Designing a privacy aware infrastructure for an Inclusive Enterprise at IBM

Master's Thesis

C.T. Steenstra

Designing a privacy aware infrastructure for an Inclusive Enterprise at IBM

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

C.T. Steenstra
born in Heemskerk, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
`www.ewi.tudelft.nl`

IBM
Center for Advanced Studies (CAS)
Johan Huizingalaan 765
Amsterdam, the Netherlands
`www.ibm.nl`

Designing a privacy aware infrastructure for an Inclusive Enterprise at IBM

Author:	C.T. Steenstra
Student id:	4089081
Email:	C.T.Steenstra@student.tudelft.nl

Abstract

The widespread adoption of computer technologies fundamentally re-shaped the way companies operate. A deluge of systems and applications now support the daily activities of employees and managers alike, thus increasing the amount, value, and sensibleness of available data. This abundance of data provides new opportunities for applications development, where more and more data is shared and reused to enable new functionalities, to unlock novel insights about the enterprise or its personnel, or to improve on aspects such as employee engagement, productivity or sociability. At the same time, data sharing poses new challenges. Data is often used for purposes that are different from the original design, and there is a pervasive need to ensure compliance with the relevant laws and third party policies. What is more, employees might find the increased use of personal data undesirable, and therefore demand proper transparency and control over their personal data.

This works tackles the technical challenges that come with the sharing and usage of personal data by enterprise-class applications, and provides *a framework for privacy aware data sharing*. In a literature survey we investigate several disciplines related to privacy, access control management, and provenance in computer systems, to determine the current state of the art and practice. The study provides the conceptual underpinning for a *novel data model* that facilitates a privacy aware way for applications to share data while still providing transparency, simplicity and control to users. The model is then implemented in a new enterprise-class platform, a *multi-tenant Software-as-a-Service (SaaS) provider* that centralises privacy and consent management related functionalities. The model and framework are then validated through interviews with IBM employees having different roles within the organisation. The quality of the resulting implementation is validated by means of a set of scalability tests, with the goal of demonstrating the actual suitability of the proposed solution in a realistic enterprise context.

Thesis Committee:

Chair:	Prof. Dr. Ir. Geert-Jan Houben, Faculty EEMCS, TUDelft
University supervisor:	Dr. Ir. Alessandro Bozzon, Faculty EEMCS, TUDelft
Company supervisor:	Drs. Robert-Jan Sips, Center for Advanced Studies, IBM Benelux
Committee Member:	Dr. Guido Wachsmuth, Faculty EEMCS, TU Delft

Preface

This report is the end product of a master thesis project in partial fulfillment of the Computer Science master program at the Delft University of Technology. This project has been conducted at the Web Information Systems (WIS) group in collaboration with the Center for Advanced Studies (CAS) at IBM Amsterdam.

Writing this preface marks an end to my journey as a student. The last few months have been a tremendous learning experience during which my perseverance, discipline and technical knowledge have been put to the test. During my studies I developed a passion and curiosity for technology and engineering which made this project a perfect fit. Through my internship I also got a glimpse of working in a large enterprise, something that I will not soon forget. I am therefore very grateful to Geert-Jan Houben, Alessandro Bozzon and Robert-Jan Sips for providing me with the opportunity to conduct this research at the WIS group and at IBM.

The completion of this thesis would not have been possible without external help, for which I would like to pay my gratitude. Foremost I would like to thank Alessandro Bozzon for his support, dedication and patience as my immediate supervisor. During the countless meetings Alessandro provided excellent guidance by giving pointers and critique while helping me stay focused on the right parts of my work. Robert-Jan Sips, my main supervisor at IBM, for his help in determining the requirements of my work, his enthusiasm in building an Inclusive Enterprise and his interests in the technical side of my project. Geert-Jan Houben and Guido Wachsmuth for their time and interest in my work as part of my thesis committee.

I would also like to thank all IBM employees, who have provided a welcoming environment for conducting my project. In particular I would like to thank Gert Geudens for his continued support and interest in my work, who always made time for guidance during implementations. Aldo Eisma for his interest in my solution and his technical input, which has given me new insights, something that I always appreciate in my endless efforts to become a better programmer. Zoltan Szlavik for his availability as a supervisor and for his helpful suggestions. My parents for their continued support during my years as a student. My older brother for setting the example as a Delft University of Technology student. My younger brother for his enthusiasm and interest in the Computer Science field and my work. Finally, my friends for providing distractions when needed.

C.T. Steenstra
Delft, the Netherlands
April 21, 2016

Contents

List of Figures

Chapter 1

Introduction

The abundance of data made available by various (Web) sources offers a lot of opportunities for novel applications and systems, or to improve existing functionalities. For instance, an application can use external social networks such as Facebook[1] to provide personalisation capabilities, or to introduce new social engagements such as content sharing. Similarly, applications that already gather data for some purpose might find additional purposes for this data, either by using it directly or by sharing it with other applications.

The use of various data sources has equal value in enterprise environments. Through the use of various computer systems different aspects of an enterprise such as employee engagement, productivity or sociability can be improved upon. For instance, external services such as LinkedIn[2] can provide insights into employee talents and experience which in turn can be used for employee project assignments. Data gathered by existing systems within an enterprise can also be leveraged for other purposes than for what the data was initially gathered for.

However, data usage is often paired with restrictions, imposed by either laws or external data providers. Furthermore, consent for data usage can be required. Thus, data usage imposes several challenges for application developers. For employees the use of their personal data can also become troublesome, especially when there is a lack of control and transparency. Consider for example an application within an enterprise that allows employees to chat with a counsellor about their well-being. An employee might be reluctant to use this application when they know their conversations are shared with the human resources department. Not disclosing such data sharing introduces ethical concerns which additionally might harm public perception of the enterprise when such usage eventually comes out.

Applications can often provide users with trade-offs in regards to the use of personal data: an application asks for additional personal data from a user, thus potentially invading their privacy, but in return the application can provide some added value. The worth of this added value could vary per individual and thus the trade-off will most likely be made differently per user. Trade-offs between privacy and disclosure are discussed by works such as [1], in which both preserving privacy and disclosure of private data are viewed as having a certain economic value. In order for users of an application to be able to make an appropriate privacy trade-off there should be

[1]`https://www.facebook.com`

[2]`https://www.linkedin.com`

adequate functionality to do so. This work mainly focuses on the technical implications of the necessity for privacy trade-offs.

1.1 Example: Enterprise chat application

Consider a simple chat application for employees within an enterprise. Such a system, although simple, provides a rich source of information that could possible be used for various purposes. For example:

1. Chat meta data can be used to get insights into communication flows between departments. These insights can be used to improve collaboration efficiency by relocating certain departments within an office building.
2. Anonymised chat messages can be used for a machine learning experiment.
3. Chat logs can be used to profile users in order to determine expertise within an enterprise. This information can in turn be used to approach employees with challenges based on their expertise.

These use cases all have different purposes and vary in terms of personal data disclosure. Some users might be willing to share their data for every use case while others rather not share their data at all. In most cases a user agrees upon a middle-ground between functionality and privacy. The added value for data disclosure in these use cases could respectively be: increased productivity, contributions to scientific research and the possibility for more challenging and engaging work. The worth of these improvements could vary per individual and thus the trade-off will most likely be made differently per user.

In the example chat application privacy trade-offs can be included in various ways. One way is to let the details of data sharing up to the application administrators. In this situation the users have little say in privacy trade-offs; they can either use the application or find an alternative. Ideally however the application would contain an overview of data usages together with an opt-in system. This way, each user of the application can decide for which purposes they would like to share their data.

Although the second approach would be a desirable solution, it does provide the chat application with the extra burden of bookkeeping. For one application the development overhead of such a solution could be manageable, but the same functionality could be useful in various other applications that require data sharing. This overhead would become even more complex when several applications would like to share data with each other.

Such a solution could be equally cumbersome for users, considering that every application will have their own privacy settings implementation. This means that privacy settings are distributed over different applications without a common interface, both in terms of GUI and vocabulary. This work tackles these challenges by designing a framework for centralised data sharing in which privacy, transparency and opt-ins are main concerns. By providing these functionalities as a single separate service developers can focus on the core functionalities of their applications while users benefit from transparent, familiar and concise privacy management.

1.2 Context

This work is conducted as part of an internship at IBM. The main use case for this work is the vision of an Inclusive Enterprise [33]. In this vision various computer systems within an enterprise work together to improve employee job satisfaction through increased engagement and overall well-being. These computer systems could benefit immensely from data sharing, but data usage should be transparent and users should remain in control over their personal data.

The framework designed in this work is a *multi-tenant Software-as-a-Service (SaaS) provider* that centralises privacy and consent management related functionalities. Multi-tenancy is defined as an architecture in which separate computer systems are unaware of the existence of other computer systems. However, in this work computer systems can still communicate with each other indirectly through the framework, thus allowing data sharing in a privacy aware manner.

1.3 Current and envisioned data sharing scenarios

To further illustrate the complications that are introduced by more sophisticated data sharing two data sharing scenarios can be described; the current scenarios and the scenario as envisioned in this work. In both scenarios there is a cloud in which server applications operate. These server applications store some data and subsequently make this data available to others through an API. There are two types of server applications; internal and external applications. Internal server applications operate within the context of the framework while external services are not. These external services can range from third party services such as Facebook, Twitter or LinkedIn to existing systems within an enterprise such as IBM Connections, an enterprise social network. Additionally there are client applications which are operated by users. These client applications communicate with server applications to interact with data.

1.3.1 Current data sharing scenario

In the current scenario, as depicted in Figure 1.1, several applications communicate with each other directly. Within an enterprise there might be protocols for such data sharing between applications, but enforcement of these protocols is tedious due to the lack of overview. This situation leads to applications that interact with different data sources freely, thus increasing the risks of privacy violations.

Data flows between applications in an enterprise have been possible for a long time. However, due to the increased presence of the Internet in our society there are new ways in which personal data of employees can be obtained. Specifically, external data services can be utilised and because of this data flows can possibly break through enterprise barriers. This means that data flows are harder to control, which introduces additional risks in terms of privacy violations, international privacy law violations or external service policy violations.

Besides the risks there are additional undesired characteristics of this scenario. For application developers implementation of data sharing can become tedious, considering the various different APIs that need to be invoked independently. This is especially cumbersome when each server application behaves differently in terms of technological details such as authentication or communication medium. Furthermore, application developers must take proper actions to ensure compliance with the established protocols with regard to data sharing. When external services are invoked directly, these protocols can be hard to enforce.

For employees this scenario is also undesirable since there is little overview of how personal data is moving between various applications. Applications could keep track of this information flow, but transparency is not enforced.

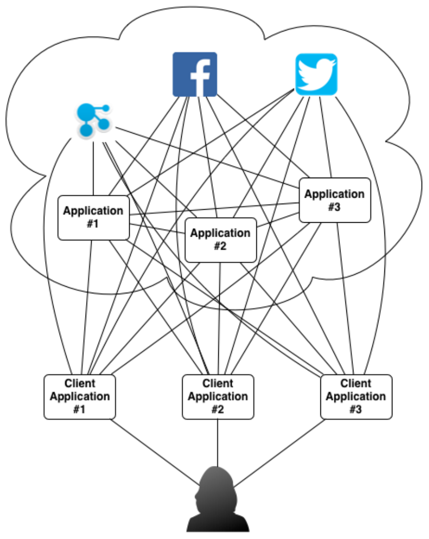


Figure 1.1: Current scenario

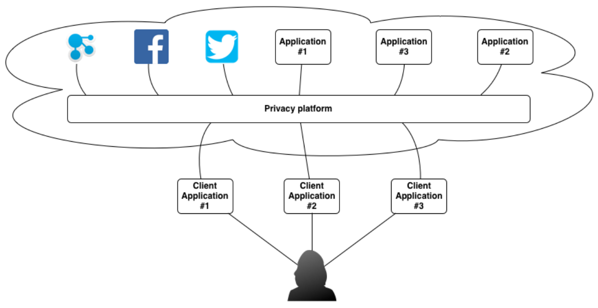


Figure 1.2: Desired scenario

1.3.2 Envisioned data sharing scenario

In contrast to the current scenario, this work envisions a scenario as depicted in Figure 1.2, where personal data management is centralised. By providing a layer between applications the enforcement of transparency and compliance with the various restrictions by applications becomes much easier. Additionally, application developers are relieved of the burden of consent management and hence developers can stay focused on building core application functionalities.

The envisioned scenario shows resemblance the Facebook graph API[3]. This API allows third party applications to interact with the Facebook social graph. Applications can ask Facebook users for certain permissions which in turn allow these applications to access this data. An overview of this is illustrated in Figure 1.3. In this model Facebook maintains personal data about users. This model can be seen as a personal data management system similar to what is envisioned in this work. In this model, access to personal data is always validated by the personal data management system, thus ensuring consent.

[3] `https://developers.facebook.com/docs/graph-api`

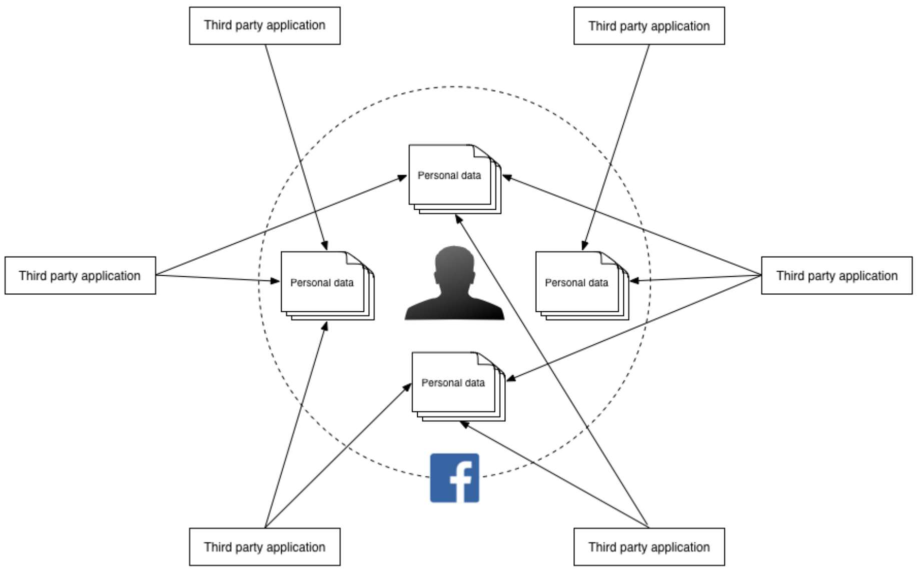


Figure 1.3: Facebook model

There is a substantial difference however between the Facebook approach and the one envisioned in this work. Specifically, in the Facebook model each application interacts with personal data maintained by Facebook. However, in a multi-tenant application environment within enterprises there is no central source of data. Rather, the innovations envisioned in this work focus on sharing of data maintained by different applications. An illustration of this concept is given in Figure 1.4. Here, each third party application maintains a custom set of personal data about users. This data should then be made available to others. Nevertheless the users can still benefit from a permission system similar to the one provided by Facebook. As can be seen in this illustration, this work envisions a system in which multiple applications can provide personal data yet data interactions still use a single privacy management layer.

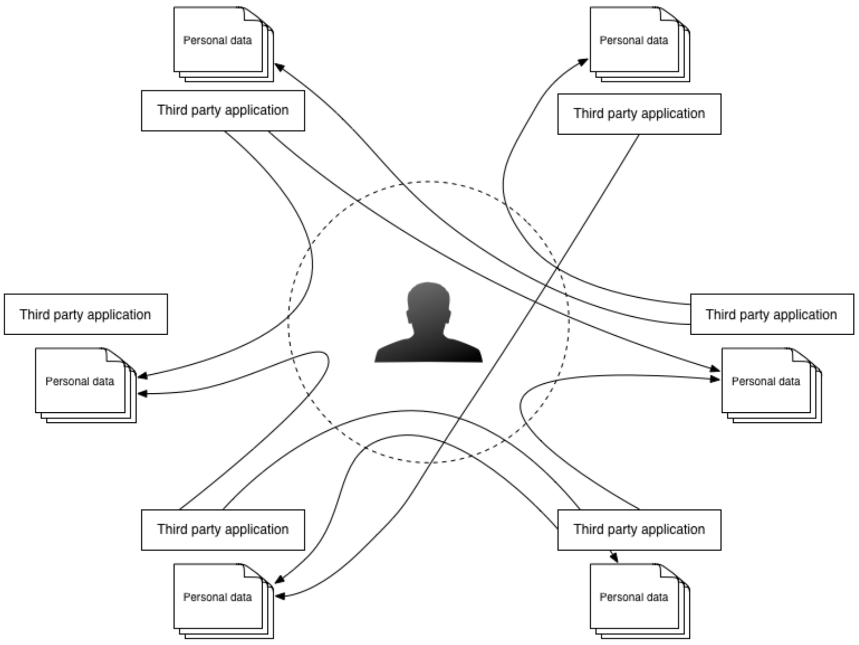


Figure 1.4: System vision

1.4 Challenges and contributions

In this work a framework is proposed for centralized privacy and access control management. This framework consists of two parts; a *model* that describes privacy related data within a multi tenant application environment and a *platform* implementation that uses this model.

Through this framework this work addresses two problems: 1) existing models relating to privacy and consent management do not incorporate all aspects desired in a centralized privacy management framework as described in the previous section. Consequently, 2) no platform implementation is available that directly implements these requirements.

There are several challenges that have been tackled. The framework requires enough expressiveness for supporting a wide variety of use cases for employee centric applications within an enterprise. Despite this expressiveness the framework is still required to be easily understandable by end-users and thus a trade-off has to be made between expressiveness and simplicity. The main implementation challenges are scalability and security.

The contributions of this work are:

- A new model extending the current state of the art in access control models in a multi-tenant enterprise application environment.
- An implementation demonstrating the approach and showing the feasibility of the system in terms of performance as required in an enterprise environment.

- A demonstration of the flexibility of the system, showing how the model supports the integration of various domain specific use cases.
- A validation of the design through interviews with IBM employees with different roles within the organization.

1.5 Methodology

The thesis starts with an analysis of the privacy and data access control requirements as stated by IBM. In this analysis the high level goals of the framework are used to define the requirements of the framework from two perspectives; 1) that of users, and 2) that of developers working with the system. These requirements are defined through use cases and system stakeholders under guidance of IBM supervisors. The results of this analysis are given in Chapter 2.

After determining the scope of this work in terms of framework requirements a literature survey is conducted to gain insights in the current state of affairs. This survey, as given in Chapter 3, provides a basis on which the framework can be built. This survey also provided additional insights in possible system requirements.

After requirement refinements the related work is re-evaluated to determine compatibility with framework requirements. Through this applicability analysis missing properties of existing work are identified. This analysis subsequently results in insights into where existing models lack and by using these insights a new model is constructed, which can be found in Chapter 4.

Using the newly constructed model the next part of this work is the platform implementation, which is described in Chapter 5. This implementation is then validated in Chapter 6 through interviews an discussions with IBM employees with different roles within the organisation as well as qualitative analysis.

Chapter 2

Problem statement

As outlined in the introduction, this work focuses on the technical implications of the requirement for personal data management functionality within applications in an enterprise environment. This chapter clarifies and states the exact requirements for a framework in which these technical implications are tackled. First, various use cases are discussed for employee centric applications within an enterprise. Secondly, system stakeholders are discussed. Then the high level goals and non-functional requirements of the framework are outlined. This is followed by a discussion of the functional requirements. Finally, multiple core concepts are discussed.

2.1 Use cases

In order to clarify the needs for some of the requirements several use cases will be discussed. Each use case entails a hypothetical computer system present in an enterprise that handles some personal employee data.

2.1.1 Office environment monitoring

A pleasant climate in the workplace can have a positive impact on employee satisfaction. The first steps in investigating the effects of office climate is setting up a monitoring infrastructure. Certain characteristics of the climate can be measured using physical sensors. Metrics can include temperature and background noise levels, which can be measured by respectively a thermometer and a microphone. For other metrics physical sensors are not sufficient, such as for example the perceived temperature. The perceived temperature can efficiently be determined by asking employees directly. The office environment monitoring computer system is responsible for collecting metrics about office climate from these different sources.

Considering the upcoming trend of wearable technologies, a more advanced monitoring approach could make use of these technologies to collect even more useful and accurate data. For instance, wearable technologies can be used to collect heart rate and core temperature data from employees. Large scale analysis of this data could provide valuable insights into office climate as well as employee well being. For individual employees these measurements can possibly be used to automatically adjust heating in their proximity. However, employees could be reluctant to provide others free access to this arguably sensitive personal data.

2.1.2 Analytics

After data has been gathered it is often not directly insightful without proper means to identify patterns and anomalies. An analytics system can be created that provides useful aggregation functionality for a wide variety of data sources. One example could be the aggregation of sensor measurements produced by the office environment monitoring system. Another example is providing a visual overview of communication flows between departments based on chat message logs, as mentioned in chapter 1. In summary, this system is responsible for converting bulk data into usable metrics and visualizations.

In order for analysis to be applicable there must be a collection of data on which to apply it. This again brings with it questions of consent and responsibility. Specifically, employees must agree with their data being used in an analytical aggregation of data. For employees this decision might be dependent on the perceived benefits offered by disclosure of personal data such as for instance a more pleasant working environment.

2.1.3 Gamification

It is desirable for enterprises to keep employees engaged. One way to achieve this is by incorporating gamification strategies into everyday tasks within the enterprise. For example, every employee that uses this application can have a score based on the number of achievements they have completed [11]. This score is subsequently used to generate a ranking of all employees. In this situation an achievement states that a specific criteria has been met.

One example of an achievement is one that will be rewarded when a healthy lunch is ordered every day of the week. In this situation the gamification system will need access to data produced by other parties, which in this example would be the restaurant's payment system. In order for an employee to be eligible to this achievement, they must first approve of the data sharing between the two parties.

Another insight that can be found in this use case involves data accessibility. Until now, only direct access to data by applications has been considered. However, in an application such as this use case it is possible that some data about an employee is also interesting to other employees. For instance, the gamification progress of a certain employee might be of interest to their colleagues. Thus, the questions in terms of consent management not only revolves around direct access to data by applications, but also access to data by different employees.

2.1.4 Social network proxies

Social networks are useful in multiple situations. There are several very popular public social networks such as LinkedIn, Facebook or Twitter, as well as internal enterprise social networks such as IBM Connections.

Within an enterprise the profile of employees on such networks can be very useful for different purposes. For example, some application might be able to make use of an employee's twitter feed for experimental purposes. Another example is the use of LinkedIn connections to determine expertise within the enterprise. In this context one system could analyze these connections and determine that a certain employee has connections with an important potential client. This employee could then be approached for a discussion about this client. By utilizing the hidden talents of employees their feeling of importance can be increased while simultaneously helping the enterprise as a whole.

Social networks can already be accessed directly by individual systems. The downside of this is that it is difficult to ensure that various systems respect the use policies of third party services. Especially in an enterprise setting such as within IBM regulations are in place that impose requirements on how data is handled. Additionally, when various systems all ask employees to link their social networks separately this might decrease participation. Rather, a single proxy between third party services would provide a lot of benefits in terms personal data management.

2.2 System stakeholders

Several stake holders can be identified that are concerned with the management of personal data. Most importantly there are *application developers* and *application users*. Applications are stand-alone computer systems that provide some functionality. These applications can consume data and produce data. Applications can provide employees or other applications access to data, possibly under certain restrictions.

For application developers the role of the system is twofold; On one hand the system should make it easier to integrate existing data sources into applications. On the other hand the system should make it easier for application developers to share their data with other application developers without overhead of a custom opt-in system.

Application users, as the name suggests, use these applications to ideally receive some direct or indirect benefits. Within an enterprise these users are employees. For these stakeholders there should be granular access control over personal data. A system providing these functionalities should convey trustworthiness and reliability so that employees feel secure in using the system to protect their personal data.

2.3 High level goals

The main goal of the framework is to make it easier for applications within an enterprise to share data in a way that respects the restrictions and obligations tied to usage of this data.

The framework aims to tackle two sides of the challenges associated with data sharing. First there is the user side, which brings challenges regarding the willingness of users to share data. Secondly the development side brings challenges regarding the overhead of development associated with data sharing functionality. The user centric challenge is tackled by providing transparency and requiring explicit opt-in before any data sharing takes place. Furthermore, a clear and familiar user interface across multiple applications is envisioned which could help building trust among users. The developer side of the problem is tackled by providing a familiar, simple and efficient interface to data sharing. By using a familiar interface for data sharing developers do not have to reinvent the wheel for every application that requires data sharing functionality.

As stated earlier the framework consists of two parts. The first part is a model that describes all data needed in a privacy aware data sharing platform. The second part is an implementation that uses this model to implement a privacy aware data sharing solution. This implementation by itself can be subdivided into an architecture and a proof of concept implementation.

2.4 Non-functional requirements

Through analysis of the discussions in the previous sections some non-functional requirements can be identified. These requirements state some points that are of high importance during system design. The following non-functional requirements are identified:

NFR.1 **Secure**
A core part of the framework designed in this work is the management of personal data. When handling this personal data it is important that personal data is only disclosed to those who are properly authorized. Thus, adequate security is core non-functional requirement.

NFR.2 **Modular**
A key characteristic of the multi-tenant application environment as present in an enterprise is the inherit modularity of these applications. The framework should therefore provide a solid base in which modularity and extensibility are key.

NFR.3 **Scalable**
The framework should account for the vast amount of applications that could possibly benefit from centralized personal data management within an enterprise. This means that the framework must be designed to be scalable.

NFR.4 **User centric**
The vision behind this work is a situation in which various applications work together to improve some aspect of a user's work experience. In this vision, the goal of these applications is mainly to provide some functionality to a user. For this reason the framework should be user centric from the start.

NFR.5 **Implemented using tools compliant with IBM guidelines**
This work is conducted as part of an internship at IBM, therefore the implementation should comply with the IBM guidelines in terms of third party tools such as open source software.

2.5 Functional requirements

The framework designed in this work can be seen as an access control framework. The authors of [18] provide a definition for the core concepts in access control frameworks. The authors define *access control* to be the model which guides the access control process. The *policy language* is defined as the syntax and semantics of the access control rules. Finally the *framework* is defined as the combination of the model, the policy language and the enforcement of the two. The authors also provide some basic access control concepts; *Subjects* are entities requesting access, *resources* are entities that require protection and *access rights* are the rights of a subject towards a specific resource. Finally, the authors define *authentication* and *authorization*. Authentication is defined as the verification of credentials and authorization is defined as the process of granting or denying access to resources based on credentials. All of these concepts translate roughly directly to the framework proposed in this work.

The functional requirements of the framework specify the exact needs that should be taken into account during framework design. The functional requirements can be grouped according to several core concepts: **authentication**, **privacy policies**, **access control**, **authorization** and **provenance**. These core concepts describe general subjects that are of importance within a privacy aware data sharing framework. The rest of this section provides an explanation, motivation and list of requirements for each core concept.

2.5.1 Authentication

The first core concept is *authentication*. Authentication is defined as the verification of credentials. Within the context of this work this has multiple use cases. First of all applications should be able to request data on behalf of a user. For example, a certain mobile application can request the Facebook profile of a user who is currently using the application. In this situation two proofs have to be given, one for the identity of the user and one for the identity of the application. Secondly, applications can request data themselves, such as would be the case in an analytics application where data requests can occur without a single user explicitly invoking it. In this situation the application has to proof that it is in fact the application it claims to be.

Generally, within the framework an *actor* will make a request to the system over some transportation medium. There are two situations in which an identity should be proven. In the first situation a non authenticated actor must proof its identity to become authenticated. This situation will be referred to as *obtaining* an identity. In the second situation the identity of an authenticated actor must be *confirmed*.

The following functional requirements regarding the concept of authentication have been identified:

AU.1 **Identifying users in different contexts**
The framework is targeted against a scenario in which multiple applications are present. In this situation users can log in through different applications. Thus, the authentication should be prepared to deal with this scenario by providing the ability to identify users in different contexts.

AU.2 **Revoking previously successful authentication attempts**
In a multi-tenant application environment there is less control for system administrators due to some of the functionality being provided by third parties. This brings additional challenges regarding privacy as these third parties can misuse data after it has been obtained. Due to this challenge it should always be possible for users to revoke previous

authentication attempts. This functionality is essential for keeping users in control of their own data.

2.5.2 Privacy policies

Privacy policies specify how data is used within an application. The main goal of a privacy policy is to exactly specify both the actual usage of data as done by an application as well as the restrictions on data usage by other applications. This concept is of importance within the framework as it provides a basis for applications to specify their data usage. For users the privacy policies are of equal importance as they provide insights into the data handling practices of applications. Through these policies users can make an informed decision on whether they want to use a certain application.

Below the exact requirements for the privacy policies as applied to this work can be found:

PP.1 **Extensible specifications of data handling practices**
Applications should be able to specify their data handling practices through privacy policies. Clearly stating data handling practices is a first step in increasing transparency towards users. This transparency consequently allows users to make a more informed decision when providing consent for data usage. These practices include information such as what kind of data is stored, how long data is stored, where it is stored and how it is stored. As apparent from these examples, a lot can be said about these practices and thus the specification language should be extensible.

PP.2 **Specification of data usage restrictions**
A goal of the framework is to facilitate data sharing among enterprise applications. A common occurrence with applications that provide data services is that restrictions are imposed on data usage. Specification of such restrictions are an additional important requirement of privacy policies. These specifications must be usable to indicate the required data handling practices. Through these restriction specifications data providers can guard against misuse of their data.

PP.3 **Describing different purposes of data usage**
A key part of the framework proposed in this work is that management of personal data should be fine-grained. This means that users should be able to exactly specify who can access their data. Besides granting specific applications access to certain data, it should also be possible to specify access to data for different purposes. This facilitates a situation in which data usage can be categorized, thus providing more transparency to users.

PP.4 **Infrastructure for dealing with policy changes**
Within a multi-tenant application environment it is highly likely that requirements change regularly. For this reason the privacy policy specifications are also likely to often change. The privacy policy infrastructure should therefore be prepared to deal with these changes in a structured manner.

PP.5 **Infrastructure for dealing with policy changes by external data sources**
When dealing with external data sources it is likely that at some point in time policies of

these sources will be updated. The resulting privacy policy specifications should therefore be able to deal with such changes in a consistent manner.

2.5.3 Access control

A key part of the framework proposed in this work is the concept of *access control*. Access control is defined by [18] as the model that guides the access control process. Within this work this concept is of great importance. The access control model describes the basic way in which access rights are determined.

The following requirements for the access control in the framework have been identified:

AC.1 **User centric access control**
In line with item *NFR.4* access control should always be user centric. Due to applications being aimed towards users the access control needs to be targeted towards a scenario in which each user will have individual preferences in terms of access control.

AC.2 **Fine-grained access control**
Access control should be fined-grained, meaning that access should be controlled on the level of individual pieces of data and also in different contexts. For instance, data to a single piece of data could be accessed with different purposes. By providing such a fine grained access control basis it is possible for users to have complete control over their data. Using this principle a solid basis can be build needed for a trustworthy privacy management framework.

2.5.4 Authorization

Authorization is concerned with the delegation of access rights. For the framework proposed in this work this concept forms a key part. Before any data is actually shared between applications there should be an explicit opt-in by the users involved. The process of providing applications with opt-ins is described by the authorization strategy.

The requirements of the authorization within the framework are as follows:

AO.1 **Opt-ins for data usage per application**
Users should be able to grant a specific application access to specific data. Data should not be available to this application without explicit consent from the user involved.

AO.2 **Opt-ins for different purposes of data usage per application**
It should be possible for applications to ask for consent for the same data for different purposes and subsequently it should be possible for employees to accept or deny consent for a subset of these purposes.

AO.3 **Revoking opt-ins at any time**
Users should be able to revoke a previously given consent at any time, and thereafter be sure that the data in question is not released under justification of the revoked consent.

AO.4 **Configuring scope of data usage**
Users should be able to configure the scope of data usage in which they are a subject. This means that they should not only be able to grant opt-ins to certain data usages, they should also be able to specify who else can access the data through these channels.

2.5.5 Provenance

Provenance is defined as the history of data. This history describes how certain data has reached its current form. This concept is useful in various contexts within the framework. First of all, the history of data can provide transparency to users. In this situation usage of data can be tracked, thereby providing users with insights into how their data is used by various applications. Another situation in which provenance can be useful is during access control. For this the main concern is that when personal data is used, additional manipulation and redistribution of this data makes the ownership question more complex. Specifically, when personal data is gathered about several individuals the main challenges becomes determining who has the right to grant access to this data. Provenance can potentially be used for solving these challenges.

The following requirements regarding provenance have been identified:

PR.1 **Viewing data usage of applications after opt-ins have been granted**
After a user has opted-in to some data usage the actual usage should be transparent. Transparency should be realized through the ability to gain insights in how applications actually use personal data.

2.5.6 Platform requirements

The concepts discussed in the previous sections define a high level grouping of the requirements. These concepts are mainly concerned with the model part of the framework. However, as previously stated another important part of the framework is the platform. This platform makes use of the model to provide the required functionalities in a multi-tenant application environment.

The following additional functional requirements of this platform have been identified:

PR.1 **Authentication via existing infrastructure**
Considering this project being part of the Inclusive Enterprise vision within IBM it should be easy for employees to work with this system. Therefore, employees should be able to authenticate with the system using their existing IBM Connections credentials. This requirement however can be abstracted for enterprise environments in general. Most enterprises already have an identity management infrastructure in place and therefore the system should not provide yet another one. Instead, existing infrastructure should be leveraged for identity management.

PR.2 **Extending available data services with application level data services**
Applications within the system should not only be able to obtain data from the system, they should also be able to provide data to other applications through a single privacy managed interface. This way, the available data to every application can be extended over time.

PR.3 **Platform provided privacy management**
The resulting platform should provide application developers with a full privacy and consent management solution, thereby reducing the needs for duplicate development of opt-in systems across various applications within an Inclusive Enterprise.

PR.4 **Infrastructure for gradually upgrading various platform components and providing dependent applications with sufficient means to handle version bumps**
As previously stated the requirements of applications are likely to change regularly. This has implications on privacy policies as discussed in item *PP.4*, but this also has implications on application inter dependencies. Due to the platform facilitating data sharing there will naturally arise dependencies between applications. Combined with evolving requirements this introduces challenges with regard to dependency management and versioning for which the platform should provide a sufficient infrastructure.

PR.5 **Receiving real time data events within the system**
Applications targeted by the framework focus on employee well being and job satisfaction. In some cases these applications might need to incorporate real time data in order for their functionalities to be useful. This is especially useful in systems such as office environment monitoring and gamification. For this reason, the resulting platform architecture should make it possible to deal with real time data flows.

2.5.7 Trust

The core responsibility of the framework is to provide data sharing transparency and control to users. It must be noted however that there is always a risk of malicious use of data by a third party application after data is released. When a user approves the use of certain data for a specific purpose by an application the platform will provide this data. What the application continues to do with the data is out of the system's control.

Although misuse of data remains a risk in the rest of this work it is assumed that applications interacting with the platform are trustworthy and act in good faith. The reason for this mainly is the fact that in an enterprise scenario these applications are developed within the same enterprise. Nevertheless there must be an accountability system in place, and future work might include a reporting system for malicious data usage.

Chapter 3

Related work

A key concept in this work is **privacy**. As stated in the previous chapter the framework described in this work is also concerned with the following concepts; **authentication**, **privacy policies**, **access control**, **provenance** and **authorization**. Each of these concepts has received a lot of scientific attention and in this chapter this related work will be discussed. This chapter is structured as follows; First, several basic principles regarding privacy are discussed that are applicable to systems that manage personal information. Then, for each of the concepts given above related work is discussed in which both the functional and non-functional requirements are also taken into account. This results in a conclusion for each concept indicating the applicability of existing work in the context of this work.

3.1 Privacy

An important part of a personal data management platform is privacy. One reason behind this is that users should feel comfortable with sharing their data, knowing that they remain in control of their data. Another reason is that there are various laws in place that impact how data can be used. Before describing how these privacy concerns can be respected this section will briefly summarize common principles of systems dealing with private information.

Within the platform a wide variety of data could potentially be exchanged. Some of this data can be more sensitive than others. Sensitive data about users is often referred to as *Personal Data* or *Personally Identifiable Information* (PII) [26]. When dealing with sensitive information there can be requirements from different sources to handle this data with extra care. Most importantly, several laws exist in different regions regarding the handling of PII. In other situations a company can have certain ethical guidelines with regards to data storage. Finally users of the system can demand certain privacy guarantees before they feel comfortable in using a system.

In [12] basic principles are described for systems that handle personal information. In these principles the *data subject* is the subject of some stored data. The principles are shortly summarized below, a complete explanation of these principles can be found in [12]:

PRIVACY.1 **Collection limitation principle**
Collection of personal data should be limited and collection of personal data should only

be done if it respects the law, is done fairly and is done with the knowledge and consent of the data subject.

PRIVACY.2 **Data Quality Principle**
Data should be relevant for their use, accurate and up to date.

PRIVACY.3 **Purpose Specification Principle**
The purpose of data collection should be stated before collection happens and further use must be limited to this purpose.

PRIVACY.4 **Use Limitation Principle**
Personal data is not disclosed or used for other purposes as stated beforehand, except when the data subject has given consent or when it is required by law.

PRIVACY.5 **Security Safeguards Principle**
Personal data should be protected by reasonable security safeguards to prevent malicious access to it.

PRIVACY.6 **Openness Principle**
There should be openness in how personal data is handled behind the scenes and it should be possible to obtain details regarding data storage and access.

PRIVACY.7 **Individual Participation Principle**
Data subjects should be able to obtain information about what data is stored about them and they should potentially be able to have additional control over their personal data.

PRIVACY.8 **Accountability Principle**
There should be a party accountable for respecting these principles.

These principles are the basis of various laws regarding privacy and thus form a basis to take into account when designing systems in which privacy is a core aspect. Although these principles are mostly applicable to how existing technologies are used as opposed to what existing technologies provide in terms of functionality, they are still useful for evaluating the resulting system.

3.2 Authentication

Within enterprises there is often an existing infrastructure for management of employee identities. This is equally true for IBM where employees have access to an account for IBM Connections[1], an enterprise social network platform. By using this existing infrastructure for employee identities there is less management overhead, as well as a lower entry barrier for employees to make use of the platform. IBM Connections provides an oAuth 2.0 [14] endpoint, which can be used to let employees obtain an identity on the platform.

[1] `https://www-03.ibm.com/software/products/en/conn`

For applications multiple authentication solutions exist. A common approach used in protocols such as oAuth2.0 [14] is to provide applications with credentials that can later be used to identify the application in the system. Another approach is to make use of asymmetric encryption algorithms. For instance, a public key infrastructure (PKI) can be set up between applications, such as illustrated in [6]. In this infrastructure each application would have a public key and a private key. Communication will then be done using verification of certificates using some asymmetrical encryption scheme. For such a PKI additional certificate management is needed in the form of Certificate Authorities (CA). These CAs maintain a trusted repository of public keys which can be used to obtain application identities.

After identities have been obtained it should be possible for actors to reuse their identity for additional queries to the platform. In this situation an identity must be confirmed. Confirming identities is a common requirement in web based systems and several different techniques are available for use with the HTTP protocol. The authentication requirements as listed in subsection 2.5.1 are mostly concerned with this aspect of the authentication technology. The next sections discuss several alternative authentication strategies. For each of these strategies the non-functional requirements of modularity (*NFR.2*) and user centrality (*NFR.4*) are not discussed, as for every strategy handling of authentication can be modularized and authentication is inherently user centric.

3.2.1 Basic and Digest Authentication

A simple scheme is the *Basic Authentication* scheme, and similarly the *Digest Authentication* scheme[20]. The basic authentication scheme requires every request to be accompanied with credentials included in the request headers. This way, the identity of the requester is verified directly at every request. Digest authentication uses a slightly different scheme in which the server provides a nonce that must be used by the client to hash the password. This hash must subsequently be sent by the client in the request and only if the server obtains the same hash using the same process on the stored password is the authentication accepted.

Functional requirements

There are several flaws with these schemes when applied to this work. First off, requirement *AU.1* can not be directly fulfilled by these strategies. These schemes require system wide credentials to be included in each request. These credentials do not include any additional information besides verifying the existence of a user. Thus, when requests are made through a certain application there is no direct way to verify the validity of the request context. This could lead to misuse of user credentials by unauthorized applications. Secondly, revoking previous authentication attempts as required by *AU.2* is very inconvenient. Since each application would use the same system wide credentials revoking access for specific applications requires changing the credentials, impacting every other application. Thus, revoking access from a single application is not directly possible.

Non-functional requirements

Both the basic and digest authentication strategies impose some concerns with regard to security as required by *NFR.1*. The main issue here is the disclosure of system wide credentials to multiple applications. This means there is no direct way to impose strict limits to access per application. Additionally, scalability, as discussed in *NFR.3*, is also impacted by these strategies. For every request the authentication information included in the request will have to be compared to the user's credentials. This imposes a challenge for scalability as this adds a latency overhead to every request. Additionally each system entry point will need to communicate with the credentials store, making horizontal scaling more challenging. In light of *NFR.5*; the basic authentication scheme is supported by IBM Connections, however, production use of this scheme is discouraged. This does however not restrict the use of these technologies directly.

3.2.2 Session Authentication

A second scheme is called *Session Authentication*. Although not directly standardized an example can be found in [13]. In session authentication, as the title suggests, the requests of an actor are executed within a session. A session starts by obtaining an identity. Subsequent requests belong to the same session. Sessions often work by storing a session ID in a cookie. When a request is made the session ID is used to look up information stored in the session. Thus, a session ID can be seen as temporary credentials. A key point with this scheme is that the server stores information related to a session. In the context of this work information such as the requesting application and the employee ID can be stored in a session.

Functional requirements

The session authentication strategy is mostly in line with the requirements. After starting a session a session ID acts as temporary credentials. Within the session details about the context can be stored safely, as required by *AU.1*. Since session data is stored on the server sessions can easily be closed afterwards, thereby fulfilling *AU.2*.

Non-functional requirements

Session authentication provides a secure way to manage authentication in different contexts. However, this strategy imposes some challenges in terms of scalability. Due to session data being stored on the server, each subsequent request in a session must interact with this session data for verification purposes. This can become challenging when entry points to the system are horizontally scaled. In this situation a solution must be provided to ensure every entry point has access to the same shared data. Multiple solutions for this problem exist, such as the use of shared session storage or the use of sticky sessions. In the first approach each entry point has access to the same data storage over the network. In the second approach requests in a certain session will always be routed the the same entry point by a load balancer. Although possible, these solutions add some complexity.

3.2.3 Token Based Authentication: JSON Web Tokens

A common authentication scheme which is also used in oAuth2.0 and similar protocols[14] is *Token Based Authentication*. In this scheme a token is issued to an actor after an identity has been obtained. This scheme is similar to session authentication in that a token is used to confirm an identity. For this scheme the same problems exist in terms of scalability. However, a solution to this problem is the use of the JSON Web Token (JWT) standard[17]. This standard describes a way to construct authentication tokens. These tokens consist of a number of *claims* and a certificate. The certificate is constructed using a server side private key. When the JWT is then used for confirming an identity the server can simply validate the certificate. When the certificate is valid, the claims will also be valid. Thus, an actor can be identified by its token using the certificate without storing any additional information on the server, simplifying the server architecture.

Functional requirements

As with the session authentication strategy the token based authentication strategy allows for keeping track of contextual information. However, due to the use of signed tokens for identity verification fulfilling requirement *AU.2* becomes somewhat more complicated. Since a token is signed using a secret key, it is impossible to revoke a token afterwards, besides by changing the secret key used to verify tokens. However, when this key is changed, this means that every other token is also considered invalid. To combat this issue a similar approach as used in the oAuth2.0 protocol[14] can be used. Instead of signing tokens that are valid for an infinite time, signed tokens are only valid for a short duration. These short lived tokens are called *access tokens* and these tokens are used to verify identities during requests. Besides these short lived tokens there are long lived tokens called *refresh tokens*. These tokens can only be used to obtain new temporary access tokens. The difference between these tokens is that refresh tokens are not signed, instead they are persisted on the server. By revoking refresh tokens requirement *AU.2* can be fulfilled.

Non-functional requirements

Both the session authentication strategy and the token based authentication strategy have some advantages in terms of security in *NFR.1*. Since session based authentication only stores data on the server it provides a quicker way to revoke authentication. With token based authentication access tokens can be misused without being revocable for a short time. On the other hand, token based authentication provides some advantages in terms of scalability in *NFR.3*. Stored refresh tokens only need to be accessed when access tokens are expired. For every other request the access token can be verified simply by verifying the signature. This signature can be cached in every system entry point, thus greatly reducing the required network round trips for every request.

3.2.4 Authentication summary

A summary of the authentication schemes discussed in the previous chapters is given in Table 3.3. Although session authentication is arguably also an option, the token based authentication provides some implementation benefits and will therefore be used.

	Basic Authentication	Digest Authentication	Session authentication	Token Based Authentication
AU.1	✗	✗	✓	✓
AU.2	✗	✗	✓	✓
NFR.1	✗	✗	✓	✓
NFR.2	✓	✓	✓	✓
NFR.3	✗	✗	✗	✓
NFR.4	✓	✓	✓	✓
NFR.5	✗	✗	✓	✓

Table 3.1: Summary of authentication schemes and their compliance with relevant requirements.

3.3 Privacy policies

There are several ways in which a policy can be specified. At its most basic form a policy specification is an informal text based policy description. Although useful for end users such a specification system is rather limited in terms of additional processing. Without a formal specification method it is hard to make systems that process policies. Another approach thus is to specify policies in a machine readable format. By using a machine readable format it is possible to build systems that provide additional functionality with regard to privacy policies. In order to improve compatibility between privacy systems it is desirable to make use of existing policy standards. The next sections will discuss some policy specification standards. Due to the similarity of the various standards in terms of the non-functional requirements the applicability in this context is discussed in a single section.

3.3.1 Platform for privacy preferences (P3P)

An example of a machine readable privacy policy specification is the Platform for Privacy Preferences (P3P)[28][10]. This platform describes a markup language in the XML format that can be used to describe privacy statements and practices in machine readable format. Within a

P3P privacy specification information such as purposes, data retention or access control can be described.

The authors of [28] describe the situation in which web servers expose a P3P policy reference file that can be obtained by a P3P user agent. The machine readable policy can then be converted to a human readable format for a user to inspect. Additionally the user agent can inspect the policy to determine potential privacy flaws in which the user might be interested in and give warnings accordingly.

Functional requirements

The main goal of the P3P standard matches the requirements as stated in subsection 2.5.2. The standard provides a vocabulary to encode privacy policies into a machine readable XML format. Through this vocabulary data handling practices (*PP.1*) and purposes (*PP.3*) can directly be described. This standard is mostly focused on specification of privacy as performed by a particular party with the goal of transferring this information to end users. However, *PP.2* states the requirement of imposing additional restrictions on data usage by other applications. Because of the HTTP based approach of the P3P standard, versioning of policies can be handled easily through cache control headers. This does however not directly provide a solution to the challenges imposed by versioning of policies when dealing with explicit opt-ins by users. When users opt-in to certain data usage, modification of policies should be handled properly with additional explicit consent of users. The same issues arise for *PP.5*, which requires sufficient infrastructure for dealing with external policy changes. Concluding, the ideas from P3P are valuable but additional efforts have to be made before all requirements can be fulfilled.

3.3.2 Global Enforcement of Data Assurance Controls (GEODAC)

In [21] a framework is proposed for describing policies regarding *data assurance* in the context of outsourced services. This policy specification framework shows many similarities with the requirements of the policies in this work. In this framework several categories of requirements are distinguished; *privacy*, *data migration*, *data retention*, *data confidentiality*, *data availability*, *data integrity* and *usage appropriateness*.

Privacy requirements are used to ensure that laws regarding privacy are met. An example given in [21] is the requirement for some data about EU citizens to be physically stored in the EU, or that data breaches are disclosed.

Data migration requirements specify conditions for data migration. The authors of [21] discuss the propagation of policies. Propagation of policies is important for ensuring that restrictions are met in the entire data life cycle. A similar concept is discussed in [26] using the term *sticky policies*. Sticky policies describe policies that are linked to data. When data is created within a system this is usually done under agreement of some policy. By keeping track of the link between the data and the policies that apply to it, additional processing can be done to enforce policy compliance. Related to this concept is provenance which will described in more detail in section 3.7.

Data retention requirements impose restrictions on how long data is stored. More specifically [21] describes the requirement to delete data at a certain point in time, as well as the requirement to notify involved parties of deletion of data.

Data confidentiality requirements are concerned with access control requirements, as well as actual storage and transmission of data in terms of encryption.

The next two categories described by [21] are *data availability* and *data integrity*. These two categories are concerned with service uptime and fault tolerance.

The final category described by [21] is appropriateness of use. In order to control appropriateness of use the authors of [21] introduce an approval process for certain actions within the platform.

Functional requirements

The GEODAC framework introduces an extensive vocabulary that can be used to describe both data handling practices (*PP.1*) and purposes (*PP.3*). Additionally, in contrary to the P3P specifications, the GEODAC framework provides additional vocabulary to deal with restrictions over data usage by third party applications. This falls in line with *PP.2*. The same challenges apply in terms of *PP.4* and *PP.5*, namely the management of different policy versions after users have granted explicit opt-ins.

3.3.3 Privacy policy extensions: Platform for Enterprise Privacy Practices (E-P3P)

Besides specifying actual policies several works focus on additional processing of privacy policies. These extensions make use of the machine readable format in which policies are specified in order to provide some additional functionality.

After a policy is specified it is not worth much if it is not enforced. The Platform for Enterprise Privacy Practices [5] aims to extend P3P by providing additional policy enforcement measures. Through E-P3P enterprises can internally monitor and enforce compliance with P3P policies as presented to customers. E-P3P introduces a terminology to describe in more detail how data can be shared between different parties. Useful concepts used in this platform are *actions*, *obligations* and *conditions*. Actions describe different ways in which data can be interacted with. Obligations describe additional duties that are paired with the access to some data. Conditions are rules that can be evaluated in order to determine if access to some data is allowed.

Another example of automated enforcement of policies is the framework given in [32][31][30]. This framework aims to provide a framework that can be used to assure compliance with various laws. These works are however more focused on the legal side of compliance with the several existing privacy laws and less on the user side of the disclosure of data handling practices.

Functional requirements

The additional features offered by the E-P3P extensions to P3P provide a more complete fulfillment of the privacy policy requirements. With these additional features, both data handling practices (*PP.1*), data usage restrictions (*PP.2*) and details about data usage purposes (*PP.3*) can be described accurately. As with the previous policy specification standards, the additional requirements of dealing with policy changes remain an issue.

3.3.4 Non-functional requirements

All of the specification standards discussed in the previous sections share common aspects in terms of the non-functional requirements. The requirement of security (*NFR.1*) is not applicable to this concept. Secondly, each of the specifications discussed provide sufficient means to specify policies on a fined grained level. This is in line with the non-functional requirement of modularity (*NFR.2*). Scalability (*NFR.3*) is not directly applicable, but when dealing with various policy versions this might become an issue. In this context scalability is mostly concerned with the maintainability of various policy versions available.

All policy specifications are constructed by service providers, this means that users have little input on the policies besides having the choice to either use a certain service or not after inspecting the privacy policy specification. Thus, in light of *NFR.4* users are not directly centrally involved in the policy process. Nevertheless, when policies are fine grained users can also decide on a fine grained level whether they accept or reject a certain policy simply by either using or not using a certain service. Finally, the *NFR.5* requirement is not directly applicable.

3.3.5 Privacy policies summary

The specifications discussed in the previous sections are fairly sufficient in terms of the functional requirements. However, none of these standards directly provide a standard solution for dealing with policy versioning in context of explicit opt-ins. Summarizing, the specification standards provide a lot of useful vocabularies that are directly applicable to this work. Arguably the versioning requirements do not fall under the responsibilities of these standards. For this reason, the rest of this work will focus less on reinventing a policy specification language but rather on how policies can be combined in a multi-tenant application environment.

	P3P	GEODAC	E-P3P
PP.1	✓	✓	✓
PP.2	✗	✓	✓
PP.3	✓	✓	✓
PP.4	✗	✗	✗
PP.5	✗	✗	✗
NFR.1	✓	✓	✓
NFR.2	✓	✓	✓
NFR.3	✗	✗	✗
NFR.4	✓	✓	✓
NFR.5	✓	✓	✓

Table 3.2: Summary of privacy policies and their compliance with relevant requirements.

3.4 Access control

The literature provides a broad spectrum of work regarding access control. These existing access control frameworks and standards deal with challenges similar to those tackled in this work and therefore related work provides a wealth of useful information. As previously stated the authors of [18] define access control frameworks as the enforcement of an access control model in combination with an access control policy language. In this section the different access control models are discussed in order to determine whether any of these models fit the requirements as given in chapter 2.

For each model the applicability in terms of the functional requirements are discussed. Additionally some of the non-functional requirements are discussed, namely the requirement of scalability (*NFR.3*) and user centrality (*NFR.4*). Security, modularity and compliance with IBM guidelines are not discussed because these requirements are largely irrelevant to the access control model.

3.4.1 Discretionary Access Control (DAC)

In the DAC model access rights are determined on an individual basis [29][18]. Furthermore, each individual requester can delegate privileges to other requesters. The DAC model can be represented by an access control matrix in which each column indicates a resource and each row a requester. Every position in this matrix states the privileges of a requester on a resource. A typical application of this access control model is in the context of operating systems [19][29].

The model revolves around three terms; *objects*, *subjects* and *actions*. Objects are the resources, subjects are the entities requesting access to resources and actions are the activities that can be executed upon objects. The matrix is defined as a set of tuples of the form (s,o,A), where s is a subject, o an object and A a set of actions.

Functional requirements

The core concept of the DAC model is in line with the system requirements. A desired attribute of this model is that access control is handled on an individual level which is in line with *AC.1*. Furthermore, the notion of actions being performed on objects is a useful abstraction. However, the model lacks slightly with regard to *AC.2*. Access control can be granularly specified on individual pieces of data, but more than this is not directly accounted for. The situation where one subject requires the same access privileges to one resource but for two different purposes is not accounted for. Secondly there is no notion of meta data such as obligations regarding the use of data. Concluded, this model provides some basic useful concepts for the system design, but the model requires some additions before it can be used in this work.

Non-functional requirements

The DAC model provides a fairly basic access control model. An upside of this is that it can be fairly well scaled in terms of determining access rights; When access rights need to be determined there should simply be an entry in the access control matrix. However, there can be a substantial administrative overhead, especially when data is shared among many individual users. Because every combination of a specific piece of data in combination with a certain user has an entry in the access control matrix, a lot of entries are required for public data. Thus, some considerations have to be made when looking at *NFR.3*. The requirement of users being centric (*NFR.4*) fits this model naturally given that access rights are determined on an individual basis.

3.4.2 Mandatory Access Control (MAC)

In the MAC model a central authority is responsible for enforcing access control based on a certain set of regulations [29]. One example of a regulation policy is the multilevel security policy. In this example each requester is assigned a certain security level which indicates to what resources the requester is granted access. A higher security level grants access to more resources. A situation where such a regulation policy is used is in the military where a rank indicates what information a person can access [18].

Functional requirements

The applicability of this model to this work is rather limited as *AC.1* requires access control to be determined on an individual level. In the MAC model however access control is centralized. Because of the user centric approach in this work there are generally no hierarchical strict subsets of access rights and thus a multilevel model is not sufficient. The second requirement (*AC.2*) requires fine grained access control over every piece of data. Although in the MAC model access to every piece of data available can be determined separately, it is not possible to specify access rights on an individual user basis.

Non-functional requirements

The MAC model provides some scalability in terms of access right lookups, although this also depends on the regulation scheme used. For the multilevel security policy such lookups only require the verification of the correct security clearance level of the requester. However, this model is contradictory to the user centrality required by *NFR.4*.

3.4.3 Role Based Access Control (RBAC)

In the RBAC model access control is enforced via roles. Within this model each user has a role which determines the exact access privileges of the user[29]. The RBAC model has received a lot of research attention, especially in enterprise environments. The reasoning behind this is that within an enterprise access control often depends on the job title of a requester. For instance, an administrative employee requires access to certain administrative systems, while a developer needs access to code repositories. This maps intuitively to the RBAC model in which a job title can be modeled by a role. Another argument for the use of RBAC as given by [25] is efficiency. The authors of this work state that identity based access control mechanisms would be severely inefficient and too complicated while RBAC requires a lesser administrative burden.

Functional requirements

The requirement of *AC.1* states that access control should be user centric. This requirement does not match naturally with the RBAC model. The same can be said about *AC.2*. Due to access rights being strictly bound to certain roles there is less focus on individual users.

Non-functional requirements

Although it is true that an identity based model does introduce a heavier administrative burden, use of a role based access control model will not solve this issue for this work. The system designed in this work requires a fine grained access control mechanism on a user level as stated by *NFR.4*. This means that access to resources for a specific application vary heavily on the users who are involved with the resources. Mapping this to the role based model will result in a large number of roles, thus eliminating the efficiency advantage. Furthermore, the use case in [25] states that users often switch roles which means that their privileges are drastically altered. In contrast, when dealing with personal data, access to this data does not directly depend on a user's roles. Besides providing no additional efficiency advantages this model does not map naturally to the identity based nature of a personal data management system. For these reasons the RBAC model does not provide many advantages in terms of scalability (*NFR.3*).

3.4.4 Attribute Based Access Control/Rule based access control (ABAC)

A more expressive model compared to RBAC is ABAC. In this model the decision to accept or deny access to a resource is based on rules over attributes of the resource or the requester [18]. Example attributes could be a requesters birth date. Rules can be in the form of boolean expression over these attributes. Note that RBAC can be modeled by ABAC by storing a role in a requester attribute and by adding rules regarding this role. Other work concerning ABAC is often focused on distributed systems. In a distributed setting identities of requesters are often unknown to resource providers. The ABAC model can encapsulate a lot of different access control models, and while access to personal data could be modeled using ABAC, scalability of the model has to be taken into account.

Functional requirements

ABAC provides a broad model in terms of access control. Using attributes both user centric (*AC.1*) and fine-grained (*AC.1*) access control can be realized. This model therefore provides a sufficient basis for this work in terms of functional requirements.

Non-functional requirements

The ABAC model provides a lot of freedom in terms of actual access control. Using the ABAC model almost any other model can be replicated through the use of different rules. This provides enough flexibility to fulfill the requirement of user centrality (*NFR.4*). The model does introduce a heavy administrative burden depending on the expressiveness allowed in rule specifications. In contrary to the DAC model the administrative burden is mostly caused by the fact that rules have to be evaluated in order to determine access rights. It could be possible to efficiently evaluate certain rules the model, but performance needs to be taken into account in this model.

3.4.5 Access control models summary

Due to the fact that both the MAC model and the RBAC model are less focused towards individual users these models are less useful in this work. In contrary both the DAC model and the ABAC model provide a basis for the functional requirements as given in subsection 2.5.3. For both models some considerations are necessary before being directly usable. Due to the limited freedom in the basic DAC model some additions are required to deal with more fine grained access control. Although the ABAC model provides a lot more freedom, there are some restrictions that must be imposed in order to preserve scalability.

	DAC	MAC	RBAC	ABAC
AC.1	✓	✗	✗	✓
AC.2	✗	✗	✗	✓
NFR.3	✗	✓	✓	✗
NFR.4	✗	✗	✗	✓

Table 3.3: Summary of access control models and their compliance with relevant requirements.

3.5 Access control policy specification languages

As previously mentioned part of a full access control solution is an access control policy language. The policy language describes the syntax and the semantics of an access control model. Several works suggest a custom policy language together with their access control solution. In this section examples of general access control languages are discussed.

3.5.1 eXtensible Access Control Markup Language (XACML)

The eXtensible Access Control Markup Language (XACML) [35] is an XML based general purpose access control policy specification language. The language can be used to describe a wide variety of access control models. The specification provides the policy language as well as an implementation architecture.

In XACML a policy consists of three main parts; the *target*, *rules* and *obligations*. The policy target can be used to specify the applicability of a policy to a request. Using XACML different logic can be applied to different parts of the request, such as the *subject* performing the request, the *resources* that are being accessed, the *actions* that are being performed or the *environment*. When a policy is applicable to a request the policy rules can be evaluated on it. A rule consists of a *target*, a *condition* and an *effect*. A rule's target determines the applicability of a rule to a request and thus is similar to the target of a policy. The condition of a rule states a boolean condition that must apply for the rule to be approved. The effect of a rule states the outcome of when the rule is approved, which is either *permit* or *deny*. Additionally a policy can specify a rule combining algorithm. This algorithm determines how outcomes of several rules are combined into a final outcome. A combining algorithm can similarly be applied to multiple policies. Finally, *obligations* can be specified that indicate certain actions that need to be performed after access is granted.

The architecture suggested for XACML consists of multiple parts. The *Policy Administration Point (PAP)* maintains the access control policies. The *Policy Decision Point (PDP)* processes policies an requests in order to determine whether access should be granted. The *Policy Enforcement Point (PEP)* serves as a central point to which requests are made. The PEP uses the PDP to determine whether a certain requests is allowed according to the policies and if so proceeds to make this request possible. Finally, the *Policy Information Point (PIP)* manages information about users within the system. This information is used by the PEP to determine whether a user should be granted access to a resource.

XACML and the suggested architecture provide a lot of functionality that can be used to enforce almost any access control model. This flexibility however does come with some downsides in the context of this work. As mentioned in [16], the complexity of security policies determines the performance of policy evaluation. Secondly, XACML has less direct focus on privacy and consent. These factors can be taken into account when designing security policies, but this leaves a lot of the overall system requirements unsatisfied initially. Finally, as also mentioned in [16], security policy updates are not directly accounted for. This is especially troublesome considering that in the system designed in this work users should be able to modify policies regarding their own data. For these reasons, this work therefore this work will instead extend on XACML.

3.5.2 Data handling policies

In [4], [3] and [2] a privacy-aware access control system is suggested. In this system access control is handled through data handling policies described in a custom policy language. The language can be used to specify who has access to what data. The model in this work steers towards an open distributed setting in which different service providers can exist. Other involved parties are service users and external parties. Both of these parties communicate with the service providers exclusively through a negation process in which disclosure needs of information can be determined. The setting is open in the sense that identities are not strictly needed in every situation, in contrast to the Inclusive Enterprise system in which identity is always required. The following paragraphs discuss useful insights found in this work.

First of all several privacy requirements are discussed that are considered key features that a privacy aware access control model should provide. The following requirements are listed: *openness*, *individual control*, *collection limitation*, *purpose specification*, *consent*, *data quality* and *data security*. All of these requirements are also desired in this work.

Secondly this work provides a useful terminology. The policy language distinguishes *actions*, *privacy profiles*, *restrictions* and *conditions*. Actions specify privacy-relevant operations. Privacy profiles is a stored set of personal information. Restrictions are divided into several different concepts: *Purposes* specify the reason why a certain access is needed. *Provisions* indicate actions that need to be performed before data is handled. *Obligations* indicate actions that need to be performed after data is handled. Finally, *conditions* indicate boolean expressions over resource or requester attributes resulting in a positive or negative access approval. All of these concepts are useful in the Inclusive Enterprise, although details regarding their use differs slightly.

3.5.3 Access control policy specification languages summary

Existing access control policy specification languages provide the means to describe and enforce various access control models. Although the specification languages discussed in the previous sections provide useful terminologies and similar implementation requirements this work focuses on designing a scalable user centric access control model. Future work could focus on testing the applicability and performance of existing tools at enforcing this model.

3.6 Authorization

Authorization is concerned with the delegation of access rights. This concept is closely related to access control but introduces some significant implications into access control models and is therefore discussed separately. In this section several key observations found in related works are discussed.

3.6.1 Administrative policies

In [29] authorization is described in the form of administrative policies. These policies dictate who can modify access control privileges. Several policies are distinguished. In a centralized authorization policy there is a single authority that modifies access rights. In an hierarchical policy administrative privileges can be delegated. In a cooperative policy authorizations need to be approved by several parties. In an ownership policy data subjects are owned and owners have administrative privileges over their own data. The decentralized policy combines features of all of the above policies.

In this work the goal is to provide users control over their data. Out of the several administrative policies given above only one policy fits this goal: the ownership model. When dealing with personal data the ownership policy is a natural fit. In this case, personal data is owned by the subject of data.

3.6.2 oAuth 2.0

A commonly used authorization framework is oAuth 2.0 [14]. This framework is described as follows:

> *“ The OAuth 2.0 authorization framework enables a third-party application to obtain limited access to an HTTP service, either on behalf of a resource owner by orchestrating an approval interaction between the resource owner and the HTTP service, or by allowing the third-party application to obtain access on its own behalf. ”* [14]

In this model an HTTP service provides some functionality. A third party application can use oAuth 2.0 to obtain limited access to this functionality. This functionality is not strictly defined, examples include invoking actions on the HTTP service or retrieving some data. A third party application can specify what functionality they would like to obtain through the *scope* of an authorization request. The scope is usually presented as a comma delimited string with keys where each key specifies a certain functionality. Upon making the authorization request the scope can be provided by the requester, the server consequently responds with a scope indicating which functionalities have been obtained. The scope model provides the basics of opt-in authorization to third party applications.

This model is a useful basis for a personal data management system, but lacks in some significant aspects. First of all, the scope model is very basic and does not formalize all aspects of the offered functionality. For instance, the reasoning behind the need for usage of a certain functionality is not directly defined in the model while this is a very desirable feature. Another key difference is the fact that within an enterprise there is no central HTTP service that provides functionalities, but rather there are multiple applications providing functionality. Setting up an oAuth 2.0 endpoint for every application would be cumbersome and against the goals of the Inclusive Enterprise system.

3.6.3 Authorization summary

There are several aspects of authorization that are needed in this work. On one hand authorization must be administered correctly. As stated earlier a natural fit for this work in terms of administration is the ownership administration policy. This means that data is owned and an owner of data has full rights to determine further decisions on authorization regarding this data. Although this provides a basis for authorization, it does not provide a full solution. In order to fulfill the requirements as given in subsection 2.5.4 additional efforts are needed.

3.7 Provenance

There are different levels to which provenance can be tracked. For instance, the Open Provenance Model (OPM)[22] can be used to describe the entire history of data within a system. At the core of this model there are *artifacts* which represent some immutable piece of data. New artifacts can be created due to actions and a series of actions is termed a *process*. *Agents* are entities that perform processes. The model itself is a directed graph in which artifacts, processes and agents are all included as nodes. The edges between nodes represent different information about provenance. The different edges are; *used*, *wasGeneratedBy*, *wasControlledBy*, *wasControlledBy*, *wasTriggeredBy* and *wasDerivedFrom*. By looking at the edges between nodes the entire history of a piece of data can be analyzed.

When provenance is stored analysis can be executed upon it for various means. For example, the provenance data can be used for access control[23][7][24][36]. In these use cases the history of some data is analyzed to determine whether access should be granted. In some of these models originators of data can specify their preferences regarding further disclosure of the data. However, this work will not use provenance to such extent. Instead, data usage can be tracked for transparency reasons as required by *PR.1*.

3.8 Conclusions

The related work discussed in this chapter combined with the requirements lead to some conclusions regarding the use of existing models, frameworks and standards. An important observation that impacts these conclusions is that the system designed in this work is mainly focused on handling personal data. It is important that users remain in control over their data and that transparency in data usage is present. For this reason a core part of the platform will focus on these aspects. This focus is realized by a number of design decisions regarding existing work.

Foremost all the privacy principles as given in section 3.1 will be used as a guideline as to what the framework designed in this work should offer and how this functionality is realized. Secondly token based authentication is used as an authentication technology.

As previously stated transparency is an important part of the platform and therefore privacy and data usage are important components to take into account. The main requirement for privacy policies within the system is that they provide enough freedom to express a variety of privacy aspects and that they can efficiently be used to show users relevant information. This means that within the core model of the platform privacy policies should be included. For this, existing privacy policy languages provide a sufficient basis and therefore can be used either directly or as an inspiration. Although these specification standards are mostly complete in terms of vocabulary a challenge will be the management of policies in a multi-tenant application environment. In such an environment a lot of dependencies between applications can exist which becomes especially difficult in combination with user consent management. For these reasons the rest of this work will mostly focus on the model surrounding privacy policies instead of on the policies themselves, as sufficient means for describing these already exist.

As an access control model a natural fit for the system is the DAC model. In this model explicit consent has to be present before data can be accessed. In contrary, other models such as MAC, RBAC and ABAC can provide access to certain parties in less direct manners. Although such an approach can provide benefits in situations in which access to data is less standardized, in a personal data management platform data will mostly revolve around users. In this situation it would make less sense for users to indirectly grant access to resources to other users. A downside with this model is that it provides a substantial administrative overhead for data that should be accessible by a lot of users. Alternatively the ABAC model can be used to combat this issue, but as previously stated scalability needs to be taken into account.

In terms of authorization users should always be in control of their own data, the ownership administration model therefore provides a convenient basis. In this approach a user can always be the owner of their data. Through the ownership administration model the user then is in charge of granting others access to this data. Additional details are needed regarding the exact functionalities in terms of authorization.

Existing access control policy languages provide insights in opportunities and requirements for such systems. However, these systems mostly focus on different use cases in which access control can be more complex than is required for this work. For instance, XACML provides the freedom to define almost any access control scheme. The framework designed in this work however aims to provide a standardized way to create user centric applications with the aim of improving some aspect of the work environment. In this scenario complex access control rules are less relevant opposed to a scenario in which sensitive business data is involved. However, future work could attempt to model the multi-tenant application environment into existing policy specification languages.

Provenance can be used for various aspects of a personal data management system. Using a provenance based access control model however makes less sense in a platform in which data is mostly focused on specific users. Using this data for access control adds a lot of complexity and thus makes the access control process in the system less transparent for users. Still provenance can play a role in creating an overall more transparent system. Keeping track of historic data might provide additional opportunities for making users aware of how their data is managed. Thus, ideas from provenance can be taken into account when designing the system, but in this work the focus is less on using provenance for actual access control.

Concluding, existing models capture a lot of different features which are all desired or required in a model that encapsulates privacy and consent management concerns. However, these models lack in terms of a unified way to couple identity management, privacy policies, consent and additional access control demands. For these models it is mostly unclear or purposefully unspecified how they can be deployed in a multi-tenant application environment such as is targeted in this work. Most of these models are also aimed towards developers instead of end users, thus providing a very broad set of functionalities that increase complications as perceived by end users. Although some of these more complicated functionalities can be of use in certain situations, they are mostly out of scope for systems as targeted in this work. For these reasons, the aim is to define a complete solution in the form of an encapsulating model that incorporates both privacy policies, application data dependencies and user consent. This encapsulating model should provide an easy to use and directly applicable solution for applications that aim to benefit employees in a large enterprise while providing transparency and control over personal data.

Chapter 4

A privacy aware model for data sharing

At the core of a personal data management system there is a data model. In this chapter the constructed data model will be explained using relevant literature as discussed in the previous chapter as well as a running example.

A basic concept in this work is the notion of **data subjects**. One obvious example of a data subject is an employee, but things such as physical sensors can also be seen as data subjects. More generally data subjects can be anything that can be described in the system.

Data subjects can interact with the platform through **applications**, which is the second core concept. Applications can for instance be mobile apps that only use the platform to retrieve data from other applications. Applications can also be data centric and not directly usable by platform users but rather by other applications. Such a data centric application would generally be an API of some sorts.

Finally the last core notion is that of **permissions**. Privacy is an important part of this work and thus users of the designed platform should have fine grained control over data access. This fine grained control is achieved through permissions which control the access to data.

From the concepts given above three sub models can be identified. First of all there is the **graph model** that describes the instantiation of and relations between data subjects. Secondly there is the **application model**. This model describes applications and the data they provide and use. Finally there is the **permission model** that describes information related to privacy and its management. Together these models describe all relevant information needed for the envisioned platform. The upcoming sections will discuss each of these subcomponents individually.

The models in this chapter are displayed in the form of UML class diagrams. This model description framework has been chosen because it provides sufficient capabilities to describe the entities within the system together with their relationships.

4.1 Graph model

The graph model broadly describes how data exists within the system. Note that a goal of this model is to provide a basis for future extensions. Thus, its aim is to provide a bare minimum set of restrictions while still providing enough information to keep track of permissions.

Data within the system can essentially be seen as an object graph. A formal description of this model is given in Figure 4.1. Data subjects are modeled by **Nodes**. Edges between nodes are modeled by **Relations**. Furthermore, each node and each relation has a type, modeled by respectively **NodeTypes** and **RelationTypes**.

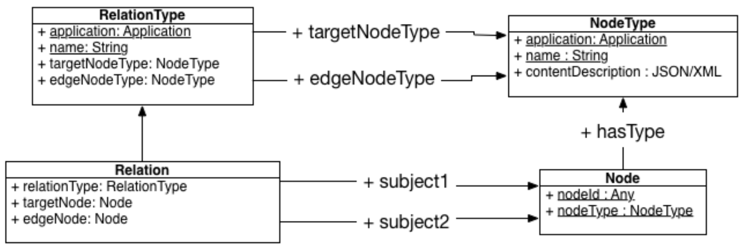


Figure 4.1: Graph model

4.1.1 Running example: Gamification platform

Although this model is rather small it can be used to model other components of the platform, namely applications and permissions. Before continuing with the model definitions a small example instantiation of this graph model is given in order to clarify the model's meaning.

As previously mentioned the running example in this report is the gamification platform. This example is used to show how the model can be instantiated to describe an application making use of the system. The gamification platform can be separated into two applications, the data component and the front end. In this separation the data component is an API exposing some functionality related to gamification while the front end provides an interface to this API for end users.

The main goal of this gamification platform is to provide gamification functionality that can easily be incorporated into other applications, thus providing a unified gamification experience that can be included into various applications with little effort. Before showing an instantiation of the graph model the gamification data model is briefly discussed.

Gamification data model

In Figure 4.2 the data model for the gamification platform is given. The components and ideas behind this model are now briefly explained before going into the graph model instantiation.

In this model users can participate through profiles. Every user making use of the platform has a single **GameProfile** describing the progress across various applications. Each user can have an additional profile per application, the **ApplicationGameProfile**, describing the progress on an application level. Each application that wants to make use of gamification can define several things, namely achievements, stats and actions. These definitions are modeled respectively by **AchievementDescriptions**, **StatTypes** and **ActionTypes**. The goal of these definitions is to provide a way for applications to keep track of subjects. Every user can have several stats as defined by an application. Users can subsequently perform **Actions** within the system that result in a modification of some stat values. Modifications of stats are modeled by **Effects**. As can be seen, each action type states what effects are triggered by it.

Besides meta data about progress applications can also define **AchievementDescriptions**. These descriptions specify the details of a single achievement within the system. This includes some general meta data, but also a list of conditions that must be satisfied in order for the achievement to be obtained. Finally subjects can obtain **Achievements** if all of their stats satisfy the conditions as described by the **AchievementDescription**.

An observation about this model is that not all entities in the gamification data model are managed by the gamification application. The user entity and the application entity are both managed by different applications. This shows a valuable insight into the graph model, namely that it imposes no restrictions on cross referencing between application specific data models in different applications.

Mapping of gamification model to graph model

The model given in the previous section highlights a useful feature of the graph model. The graph model itself is very broad, the reason for this being that it should not impose much restrictions on the actual structure of application specific data. For the gamification example this means that a model can be defined that is sufficient to describe application logic without any impairments of the graph model. The graph model can then be used to describe this application specific model on a higher level of abstraction.

The resulting graph meta data is shown in Table 4.1 and Table 4.2. This mapping highlights that relations in the graph model are always directional. It should also be noted that the graph model instantiation does not have to be completely representative of the internal application model state. This means that applications can maintain additional relations within the system without incorporating this information in the graph model. The goal of the graph model is not to provide a complete picture of application data state, but rather to provide sufficient means to reason about access to data in a unified manner.

Instantiation of the graph model

Using the graph model within the gamification context results in a graph as depicted in Figure 4.3. In this graph every instance from the gamification data model is described by nodes while relations between instances are described by edges. For instance, *Employee #1* is a node, as well as all of its profiles. Although this graph model by itself is not spectacular, it provides a very expressive model in terms of data and relations that is a good fit for the platform designed in this work considering the required extensibility. The next sections will make use of this graph model to define application meta data and permissions.

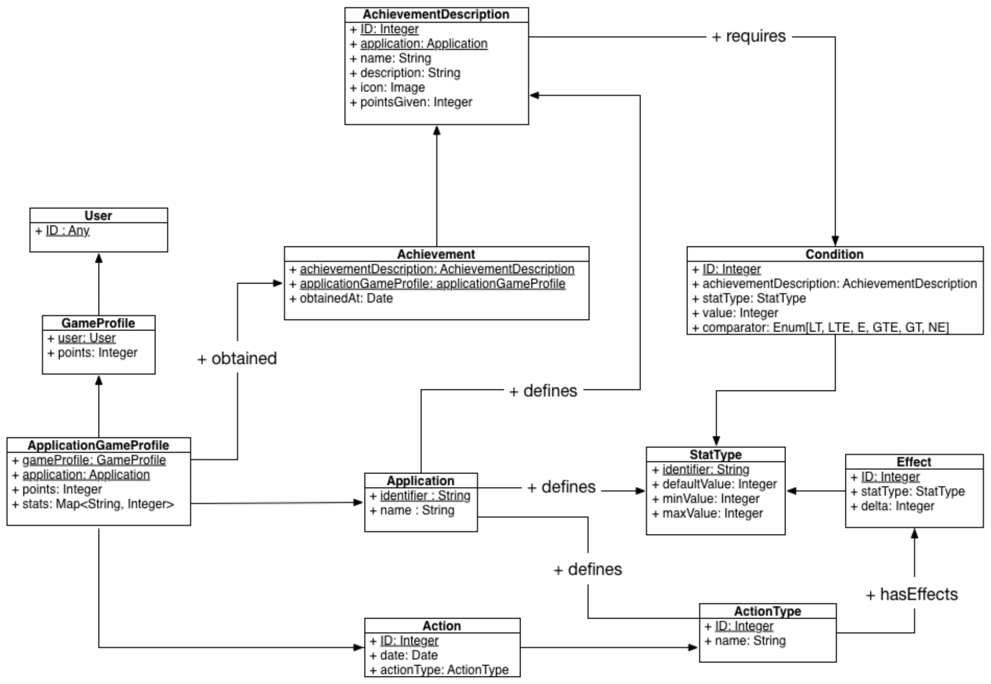


Figure 4.2: Gamification data model

Name
GameProfile
ApplicationGameProfile
AchievementDescription
Achievment
Action
StatType
Effect
ActionType
Condition

Table 4.1: Node types

Name	Target node type	Edge node type
references	GameProfile	User
partOf	ApplicationGameProfile	GameProfile
obtained	ApplicationGameProfile	Achievement
performed	ApplicationGameProfile	Action
references	ApplicationGameProfile	Application
references	Achievement	AchievementDescription
defines	Application	StatType
defines	Application	ActionType
defines	Application	AchievementDescription
references	Action	ActionType
hasEffect	ActionType	Effect
references	Effect	StatType
references	Condition	StatType
requires	AchievementDescription	Condition

Table 4.2: Relation types

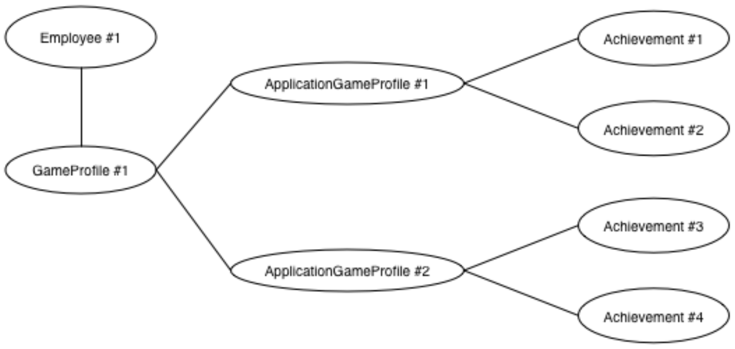


Figure 4.3: Gamification graph model instantiation

4.2 Application model

In this section the model describing applications is discussed. The model given in the previous section allows for representing data within the platform as a graph. In contrary, the model discussed in this section, as given in Figure 4.4, describes how applications interact with this graph.

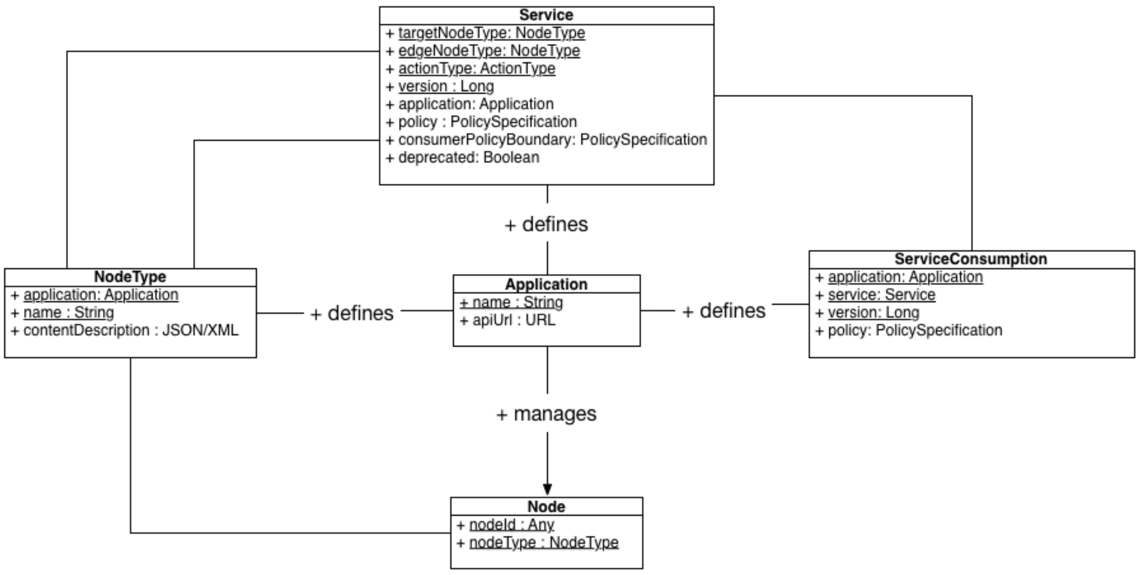


Figure 4.4: Application model

4.2.1 Applications

As previously mentioned the platform consist of several **Applications**. An application is a standalone piece of software that interacts with the graph model. Applications can define node types. When an application defines a node type this means that the application declares that it will manage nodes with this type.

4.2.2 Services and ServiceConsumptions

Interaction with the graph is done through **Services**. A service can be used to interact with a subset of the graph model. A service is provided by an application. Services are uniquely identifiable by the *targetNodeType*, *edgeNodeType*, *actionType* and *version*.

One of the node type specifications in a service can both be omitted, although a service without any node types is invalid. This leads to three valid service types:

targetNodeType	edgeNodeType	Description
Present	Omitted	Targets an instance of the target node type
Omitted	Present	Targets a public listing of the edge node type
Present	Present	Targets a subset of nodes of the edge node type which are connected to an instance of the target node type through some relation

The **ActionType** of a service describes what kind of interaction with the graph model is offered. Four action types are available: *create*, *read*, *update* and *delete*. Finally, the *version* property allows for versioning of services. Versioning will be discussed in more detail in the upcoming sections.

An important observation about services is that the application is not part of the unique identifier of a service. Rather, the application allowed to provide a service is implicitly determined by the target node type and the edge node type. This limitation is added so that applications can only provide services that interact with node types which they are managing. More simply put this means that an application can only create, read, update and delete nodes that the application manages. For the three different service types this means the following:

targetNodeType	**edgeNodeType**	**restriction**
Present	Omitted	Service can only be offered by the application that defined targetNodeType
Omitted	Present	Service can only be offered by the application that defined edgeNodeType
Present	Present	Service can only be offered by the application that defined edgeNodeType

Besides providing services an application can also consume services of other applications. The desire to consume a service is modeled by a **ServiceConsumption**. Specifying service consumptions allows for conveying data needs of applications to platform users.

4.2.3 Policy specification

An important part of the application model is the specification of policies. Policies are important for both developers and users. For developers these policies form guidelines that must be adhered to when using services provided by other applications. For users these policies give insights in how applications use their data. Several works focus on formally describing policies. In these works several useful aspects can be found that apply directly to the requirements of this work. In this section these relevant aspects, as also discussed in section 3.5, will be recapped.

Policy goals

As previously mentioned there are two main ways in which policies are used. Before detailing the policy specification scheme the main goals of these different uses are discussed. Within the application model these two different uses are identified by service policies and service consumption policies.

For services the policies describe two things. First of all they describe details about how data is handled by the service provider. This includes information such as how data is stored, for how long data is retained or whether any encryption or anonymization scheme is used. Secondly, the policy describes restrictions on how the service can be used by other applications. For instance, a service can require applications to not store any data retrieved from the service permanently. Additionally the service can impose limitations that are less technical, such as limitations on the purpose of using the service. For instance, a service can specify that it should not be used for any other purpose than scientific research.

For service consumptions the policy describes roughly identical information. However, instead of imposing limits on service usage, service consumption policies describe how data is actually used. Note that these policies should at least adhere to the restrictions imposed by the service being consumed. Secondly, for service consumptions these policies also describe the purpose of the service usage.

Concluding, three different components of the policy specifications can be identified:

- Data handling descriptions
- Service usage restrictions
- Purpose descriptions

For developers the most important components are the restrictions on service usage. For users the most important part is the purpose description, while the data handling description can also be of value. The next section will discuss these different components of the policy specification and similar ideas found in the literature.

Data handling descriptions and service usage restrictions

The data handling description component of the policy specification model must be able to express how data is handled within systems on a concrete level. As discussed in subsection 3.3.2 [21] proposes a framework for describing policies regarding *data assurance*. This policy specification framework shows many similarities with the requirements of the policies in this work. In the framework several categories of requirements are distinguished; *privacy*, *data migration*, *data retention*, *data confidentiality*, *data availability*, *data integrity* and *usage appropriateness*. Most of these categories can be applied to both data handling descriptions and service usage restrictions. On one hand a policy can describe what is minimally needed to adhere to it, on the other hand a policy can specify what is actually done. The requirements found in this framework are outlined below to show their relevance to this work in the application model context.

Privacy requirements are used to ensure that laws regarding privacy are met. An example given in [21] is the requirement for some data about EU citizens to be physically stored in the EU, or that data breaches are disclosed.

Data migration requirements specify conditions for data migration. This concept is less useful in this work, considering that data migration between applications can only occur through service consumptions with explicit consent from those involved. However, some useful insights can be obtained from [21] regarding data migration. Namely, the authors discuss the propagation of policies. Propagation of policies is important for ensuring that restrictions are met in the entire data life cycle. This concept is similar to provenance in that when data is propagated through the system, each actor over this data should adhere to the policies applied to it. For example, if one service restricts where its data is stored, then other services that consume this data and subsequently make this data available in some other form should also take into consideration these initial restrictions.

Data retention requirements impose restrictions on how long data is stored. More specifically [21] describes the requirement to delete data at a certain point in time, as well as the requirement to notify involved parties of deletion of data.

Data confidentiality requirements are concerned with access control requirements, as well as actual storage and transmission of data in terms of encryption. Within this work access control is already a core part of the permission model. Encryption requirements however are a useful addition to the capabilities of the policy specification framework. Furthermore, an additional part of the policy specification can be concerned with anonimization of data, specifying in what manner data can be linked to the original data subject.

The next two categories described by [21] are *data availability* and *data integrity*. These two categories are concerned with service uptime and fault tolerance. This is especially relevant in this work because it is concerned with outsourced services. When multiple external parties depend on each other it is desirable to have agreements in place regarding these categories. A similar situation can be found in this work, since multiple separate applications can depend on each other. Therefore agreements about data availability and integrity can also be of value.

The final category described by [21] is appropriateness of use. In order to control appropriateness of use the authors of [21] introduce an approval process for certain actions. A similar concept has been introduced into the application model, in which the use of services targeting only an edge node type require explicit permission of the owners of the application providing the service. This additional boundary on service usage, in addition to the default consent boundary imposed by the permission model, allows for limiting use of certain services to approved applications.

Conclusions

When looking at the examples given in the previous section it becomes apparent that there are different properties that are specified by a service policy. Instead of providing an exhaustive list of these properties it will be more useful to provide a policy framework that can be extended in the future with new properties. Some example properties, as found in the previous section, are:

- Geographical storage location
- Disclosure of data breaches
- Policy propagation
- Data retention

An observation about these properties is that each property can have a value to describe part of a policy. Furthermore, each property can be described as having an ordering towards another value for that property, namely as being stricter or as being less strict. In this ordering the less strict value will be called the boundary value. A value is contained by a boundary if it is equal or less than the boundary.

This observation is useful because it allows for one policy to be defined as the boundary version of a policy. In the application model context this means that a service can specify a boundary policy to which consumers must adhere. Thus, each consumer has a policy describing the use of the service and this policy is always contained by the boundary policy of the service.

To illustrate this concept further, consider the property data retention with a value of 30 days. A policy in which the data retention is 30 days or lower is a valid stricter version of the boundary policy. When defining new properties in the policy framework there should be an accompanying process to determine whether one value fits within the bounds of another value.

For the example properties given above this would mean the following;

- A geographical storage location is within the bounds of another if it is contained by it. For instance, if the storage location boundary value is ”The EU”, then the value ”The Netherlands” would be contained by the boundary.
- A value for disclosure of data breaches is a boolean, thus if the boundary is true, than the value must also be true. If the boundary is false, the value can be either false or true. The same holds for policy propagation.
- The data retention property is a numeric value indicating a duration. Given a boundary value for this duration, the property value must be equal or less than this value in order to satisfy the containment property.

A formal description of such a policy specification model is depicted in Figure 4.5.

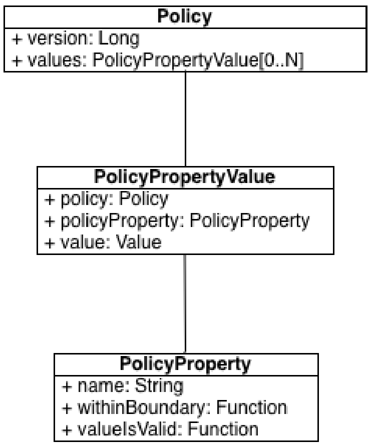


Figure 4.5: Policy specification model

Purposes hierarchy

Purposes are an important part of the policy specification as they provide end users with insights into why service access is desired. Before going further into how purposes can be described it should be noted that the purpose is essentially the same as any other property in the policy specification. Nevertheless it is discussed separately because of its importance.

A purpose describes why access to a service is desired. A purpose can be described in numerous ways, first of all it can be a simple textual and human readable description. Although a human readable description is useful, a more formal approach is desired. Using a formal approach allows for reasoning about purposes in a more concise way while also allowing for automatic processing in the same manner as other policy properties. In the literature a common approach to modeling purposes is to describe them in a hierarchy. In this hierarchy the parent relation means an abstraction of a purpose. Using this model the containment property equates to a node being in the subtree of the hierarchy where the boundary node is the root. An example purpose hierarchy is given in Figure 4.6.

A final note on the purpose hierarchy model is that this model also allows for reuse of service consumptions for multiple purposes. Consider the case where an application requests access to a service on behalf of a certain user with the purpose of scientific research. If this user gives consent to the service consumption, then the application can use the service consumption consent for any scientific purpose. Access to a service under such a general abstraction requires less consent in the future, but provides less insights to the user about how the service is used. It should therefore be a best practice for applications to provide concrete purposes when requesting service access through service consumptions.

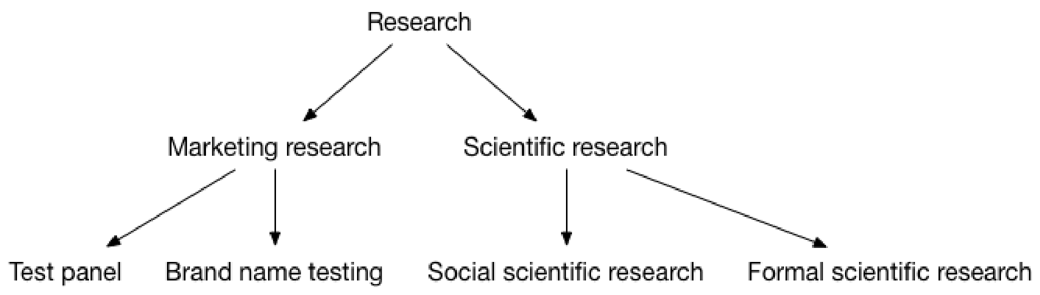


Figure 4.6: Example purpose hierarchy

4.2.4 Versioning

Within the application model multiple standalone applications exist, each of which can be developed separately. Before going into the versioning itself an important restriction is that both services and service consumptions within the application model are immutable. This means that they can not be modified after they have been declared. This immutability is essential for keeping track of permissions, since the immutability prevents policies from being modified after they have been declared. Thus, when a user agrees to a policy, this policy can not be modified afterwards.

When services are updated they can either behave differently or specify different policies. This introduces problems when it comes to dependencies between applications. To allow for a concise and maintainable system versioning is included in the application model; both services and service consumptions can specify a version. For services a version bump can be done to either indicate an updated policy or to indicate an updated service interface. For service consumptions a version bump only indicates an updated policy. Services can also be deprecated, which indicates that the specific version will no longer be available in the future. Future work might introduce more elaborate versioning such as Semantic Versioning [1].

4.2.5 Running example: Gamification platform

In the previous section the gamification model has been discussed. In order to allow applications to make use of this gamification framework it has to be exposed through services. These services each have a policy description as identified in the previous section. In the next sections a description of these services will be given to demonstrate how the application model can be instantiated.

In the gamification framework a separation can be made between two types of services. On one hand there are services that will be used by applications in order to interact with the platform. The main point of these services is to expose information about achievements and to handle user data. On the other hand there are services that allow for managing achievements. These management services can be used by application owners to set up the gamification experience within their application.

The services focused on interaction have multiple goals; First they should expose relevant information about achievements to inform users about what achievements they can possibly obtain. Secondly the services should expose user progress. Finally the services should allow for storing user progress. The following interaction services are identified:

- Retrieving all achievement descriptions for an application.
- Retrieving a user's game profile, or creating one if it does not exist yet.
- Retrieving a user's application game profile for a specific application, or creating one if it does not exist yet.
- Publishing actions as executed by a specific user.

This leads to the following service description instantiations:

Action	**Target**	**Edge**	**Description**
Retrieve	Application	Achievement description	All achievement descriptions for an application
Retrieve	User	Game profile	Retrieve a user's game profile (or create one if it does not exist)
Retrieve	User	Application game profile	Retrieve a user's application game profile (or create one if it does not exist)
Create	User	Action	Create an action of a specific type as performed by a user

The following management services are identified:

[1] `http://semver.org/`

- Creating and deleting stat types, achievement descriptions and action types for an application.
- Creating and deleting conditions for an achievement description.
- Creating and deleting effects for an action type.

This leads to the following service description instantiations:

Action	**Target**	**Edge**	**Description**
Create	Application	Stat type	Create a new stat type for an application
Delete	Stat type	None	Delete an existing stat type
Create	Application	Achievement description	Create a new achievement description for an application
Delete	Achievement description	None	Delete an existing achievement description
Create	Application	Action type	Create a new action type for an application
Delete	Action type	None	Delete an existing action type
Create	Achievement description	Condition	Create a condition for an achievement description
Delete	Condition	None	Delete an existing condition
Create	Action type	Effect	Create an effect for an action type
Delete	Effect	None	Delete an existing effect

The service descriptions given above provide the interface for interacting with the gamification framework. These services can in turn be used by various applications in order to make use of gamification functionality. This will especially be useful for the general purpose interaction services. On the other hand, the management services should minimally be used by a gamification management application.

4.3 Permission model

The purpose of the system designed in this work is to allow easy data reuse between different applications within an enterprise. The previous models allow for describing data present in the system together with their relations, as well as the ways in which applications interact with this data. The permission model, built on top of these two previous models, is subsequently used to keep track of consent of data usage. The next sections describe the details of the permission model.

4.3.1 Target nodes

As was mentioned in the previous section a service can specify a target node type. When such a node type is specified this means that any invocation of the service targets a specific node, referred to from now on as the **target node**.

4.3.2 Authentication context

Before going into the permission model itself some details about authentication are necessary. The main point of interaction with the platform is an API. This API lets applications access services of other applications. When another service is invoked, this is done by a certain application. Furthermore, the invocation is executed by some actor. In other words, an application can invoke a service on behalf of an actor. The actor is described by a *Node*. Together these two values form an *AuthenticationContext*, as shown in Figure 4.7.

The authentication context model bears resemblance to oAuth2.0 in that for oAuth2.0 a service can be invoked by a user on behalf of an application or by the application itself. The first case can be modeled by a context in which the node is the user and the application is as specified. The second case can be modeled by a context in which the node is the node describing the application and the application is as specified. This model thus provides a more general model of the oAuth2.0 authentication context.

Finally, during requests the authentication context also implicitly describes a service consumption in which the application corresponds to the application of the authentication context and the service corresponds to the service being targeted.

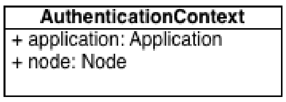


Figure 4.7: Authentication model

4.3.3 Permissions

A **permission** states that a target node within the platform has agreed with a certain policy. The permission model is outlined in Figure 4.8. There are two types of permissions, **ServicePermissions** and **ServiceConsumptionPermissions**. Whenever a service invocation is performed there must be an explicit valid permission for either the service itself (ServicePermission) or for the service consumption (ServiceConsumptionPermission). Properties of permissions and the distinction between these two types are discussed in the next sections.

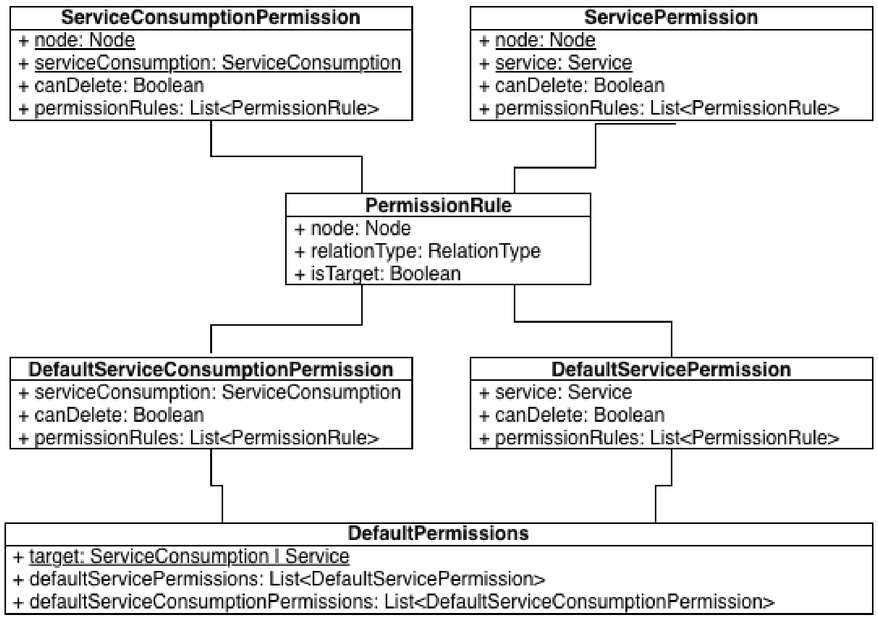

ServiceConsumptionPermission
+ node: Node
+ serviceConsumption: ServiceConsumption
+ canDelete: Boolean
+ permissionRules: List<PermissionRule>
ServicePermission
+ node: Node
+ service: Service
+ canDelete: Boolean
+ permissionRules: List<PermissionRule>
PermissionRule
+ node: Node
+ relationType: RelationType
+ isTarget: Boolean
DefaultServiceConsumptionPermission
+ serviceConsumption: ServiceConsumption
+ canDelete: Boolean
+ permissionRules: List<PermissionRule>
DefaultServicePermission
+ service: Service
+ canDelete: Boolean
+ permissionRules: List<PermissionRule>
DefaultPermissions
+ target: ServiceConsumption | Service
+ defaultServicePermissions: List<DefaultServicePermission>
+ defaultServiceConsumptionPermissions: List<DefaultServiceConsumptionPermission>

Figure 4.8: Permission model

PermissionRule

A permission has a list of **PermissionRules**. A *PermissionRule* describes a condition. This condition can be applied to a service invocation and will yield either *true* or *false*. As previously mentioned each invocation has a certain authentication context, the node within this context is referred to as the *authenticated node*. A permission rule yields true if the authenticated node has a relation of the specified type to the specified node.

The purpose of permission rules is the following; Whenever a service is invoked there is an actor which triggered the invocation; the node as specified by the authentication context. The permission rules can be used to grant multiple actors access to the service for a certain target node.

To clarify this further consider this example; Suppose there is a service that has the GameProfile as defined by the gamification application as its target node type. Now consider the game profile of a certain user. Several different nodes should be able to access this game profile, the most important one being the user that is being described by the game profile. Additionally the game profile could also be made available to friends of the user. Using rules a permission can be constructed that grants these two types of access.

ServicePermissions and ServiceConsumptionPermissions

The two types of permissions, as their names suggest, target respectively services and service consumptions. The distinction between these types of permissions is that a service permission grants any consumer of the service access to the service. A service consumption permission on the other hand only grants access to a specific service consumption. Thus, a ServicePermission is a much broader permission than a ServiceConsumptionPermission.

The reason for this distinction is that it should be possible to grant permission for any invocation of a service without specifying a service consumption. For instance, consider an application that allows users to upload images. Public images should be accessible by any application. Thus, it would be impractical to only allow service consumption permissions to be specified, because in that case a new permission must be granted for each application. Using service permissions this can be avoided. In other situations, such as the GameProfile service from the previous section, a service permission is not desirable because a user most likely wants to limit access to the service to gamification related purposes.

4.3.4 Permission deletion

A permission's *canDelete* property indicates whether the permission can be deleted. Permissions that cannot be deleted can be used to require certain permissions to always be available so that the system can function properly. Without such permissions it could become possible for nodes to exist that can not be interacted with at all.

4.3.5 Administrative policy: Ownership

The permission model provides the ability to describe permissions being granted by nodes. Another important part of the permission model is the administrative policy as discussed in subsection 3.6.1. To recap, the administrative policy dictates who can modify or access privileges. In the permission model this means that the policy dictates who can modify permissions. As was previously stated the ownership policy is used as an administrative policy.

The ownership policy implies that every node has an owner, and every node can modify permissions of itself or of the nodes it owns. This ownership relation is one-to-many, meaning that every node has exactly one owner, but it can also be an owner of multiple other nodes itself. Ownership can not be circular, meaning that it is impossible for a node to be recursively owned by itself. Some nodes inherently do not have an owner, which is represented by a node that is an owner of itself, referred to as *self owned* nodes. An example of self owned nodes are users.

Ownership can be described using the graph model, more specifically by a core relation type; the **ownerOf** relation type. Although the ownership model is rather simple compared to alternatives, it greatly simplifies the administrative policy management.

4.3.6 Edge node type

The previous sections mostly described services where there is only a target node type. For these services the permission requirements are obvious; the permission should be granted by the target node. For services with an edge node type this decision is less obvious.

These other types of services should return a collection of nodes of the edge node type. A problem for these services is that there might not be a single owner involved. Thus, for permission checking, several alternative solutions exist:

1. Do not check any permission.
2. Check the permission of every node returned by the service and filter out any node for which no valid permission exists.
3. Let applications also keep track of permissions.

These alternatives are all undesirable for the following reasons.

1. Not checking any permission at all is obviously not desired since it defeats the purpose of the platform.
2. First of all checking multiple permissions can put a huge burden on the platform, since significantly more permissions have to be evaluated for single service invocations. Secondly and more importantly, this means that the platform should do deep packet inspections in order to determine which nodes are queried. This is highly undesirable since this imposes restrictions on the data format. Rather, the platform should only be concerned with managing nodes and permissions and allow services to specify the data format without being restricted by the platform.
3. This alternative moves the burden of permission checking to applications, which is highly undesirable as well for various reasons, the main one being that it also defeats the purpose of the platform.

Service restrictions and conventions

In order to combat this issue the following restriction is introduced to services: A service that has both a target node type and an edge node type can only exist if there exists an ownerOf relationType between these node types.

Furthermore, services with both a target node type and an edge node type should adhere to the following conventions:

- When the service is invoked with the create action type, this indicates that the target node is an owner of the newly created node of the edge node type.
- When the service is invoked with any other action type, only nodes for which the target node is the owner are targeted.

Letting these services adhere to these conventions allows for making the following assumption: when a service is targeted that has both a target node type and an edge node type the result only contains nodes for which the target node is the owner. Thus, only one permission has to be checked, namely the permission as granted by the target node. Note that this would still respect the ownership model since the target node has the authority over all the returned results.

For services without a target node type there is no single node to get permissions for. Instead, the following convention is used for services without a target node type:

- When the service is invoked with the create action type, this indicates that the newly created node is self owned.
- When the service is invoked with the read action type, a public listing of the nodes with the edge node type is returned. The fact that these nodes are publicly available should be included in the policy for the service that created the nodes.
- The update and delete action types for these services do not make sense and should be avoided.

Furthermore, instead of a target node being required to grant permission, for these services a permission should be available from the application providing the service. The reason for this requirement is that some services should not be available for every other application. For instance, consider an application that is concerned with users and which provides a service with the create action type that has no target node type and the User node type as its edge node type. Thus, this service allows for creating nodes of the User node type. It should not be possible for any other application to create new users other than the one provided by the users application. Thus the users application should be able to restrict the service usage.

Service restrictions and conventions consequences

The restrictions and conventions given above are not without consequences. The model with these restrictions does not allow for services with both a target node type and an edge types to provide overlapping result sets for different target nodes, the reason for this being that there can be only one owner. This also means that there can not exist many-to-many relations directly. However, this restriction is easily avoided by using denormalization.

4.3.7 Default permissions

Another important part of the permission model are **DefaultPermissions**. Default permissions specify various permissions that are created by default when a node is created. There is a slight overlap between policies and default permissions in that a service has both a single immutable policy as well as a single immutable DefaultPermissions instance. However, not every service can specify DefaultPermissions, this is only available and logical for services with the create action type.

The reason default permissions are specified as part of a service is that it is part of the terms as accepted through permissions. Whenever a permission is created this permission states that the policies are accepted as well as the default permissions being created.

4.3.8 Running example: Gamification platform

There are several situations in which correct permissions must be in place in order for the gamification platform to be usable. Through the permission model access to different parts of the data model can be regulated in a structured manner.

The permission model imposes restrictions on data access through granular permissions. For the gamification platform this has the implication that some data must be properly initialized before anyone can make use of the system. When describing the permissions it is assumed that there are three parts of the gamification platform. First there is the *Gamification API* that exposes services for interacting with the gamifcation data model. Secondly there is a client application that provides an interface for application owners to manage gamification meta data, which will be referred to as the *Gamification management application*. Finally there is a user centric client application that allows end users to interact with the platform. This final application will be referred to as the *Gamification dashboard* and provides functionality such as an overview of achievement progress, public user profiles and leaderboards.

Below is a list of permissions that should be taken into consideration when implementing the applications given above. The permissions are mostly applicable to the client applications since they make use of services provided by the gamification API.

- Reading gamification meta data
  - Permission desired by:
    - Application owners using the gamification management application
    - Everyone using the gamification dashboard
    - Third party applications providing gamification integration
  - Permissions granted by:
    - Application owner
- Creating and modifying gamification meta data
  - Permission desired by:
    - Application owner using the gamification management application
  - Permissions granted by:
    - Application owner
  - Newly created permissions:
    - Access to newly created data by gamification management application
    - Exposing gamification meta data publicly on gamification dashboard
- Updating achievement progress
  - Permission desired by:
    - Third party applications providing gamification integration
  - Permissions granted by:
    - Owner of an ApplicationGameProfile

4.4 Combining the models

The three models given in the previous sections are all focused on specific responsibilities. Together these models provide an almost complete picture of the resulting platform. However, another important part is how these smaller models are combined into a single high-level model.

The three models given in the previous sections map to the different layers that are present in the resulting framework. First of all the graph model describes available data in the system; the data layer. Secondly the application model describes how the data layer is manipulated; the application layer. Finally, the permission model puts restrictions on the interactions between the application layer and the data layer. In this layered framework the permission model acts as a barrier between applications and the graph. An overview of the combined models is given in Figure 4.9.

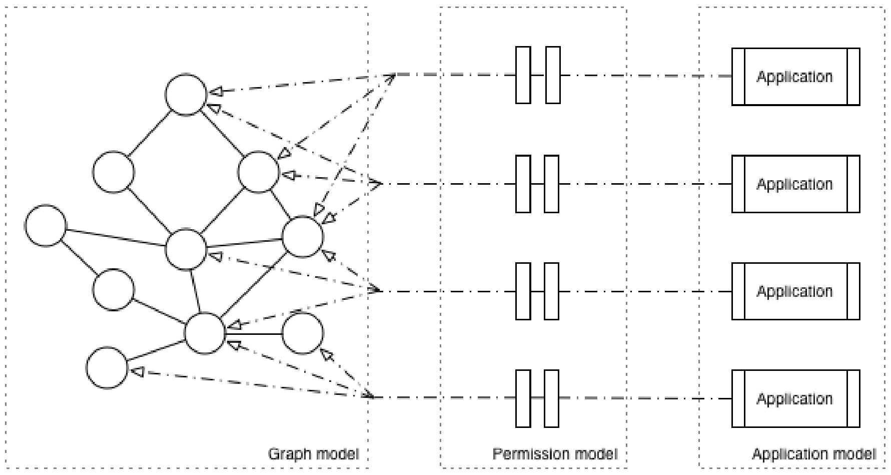


Figure 4.9: Combined models

Chapter 5

A privacy aware platform for data sharing

In this chapter the implementation strategy is outlined. First the overall system architecture is discussed including the core technology stack. Secondly the core system components are discussed together with a motivation behind the technologies used for each implementation.

5.1 System architecture

The model given in the previous chapter allows for describing data interactions between applications. Using this single model for multiple applications has several advantages, the main one being that it will be easy to reuse services. An example of a reusable service is the main motivation for this system, namely a privacy service that grants users fine grained control over the use of their data. Additionally, things like analytics can also be implemented at a single point and subsequently be integrated in various applications.

Using the application model the system can essentially be seen as an aggregator for various APIs. This results in an architecture in line with the Service Orientated Architecture (SOA) pattern. Although a SOA can rapidly become more complex than a traditional monolithic application it provides a lot of benefits that fit the requirements of the system designed in this work.

Because of the modularity inherit to this architecture applications can easily be developed separately. These applications are only coupled through the platform, which they use as a gateway through which they communicate. Separate applications can be upgraded individually without causing direct issues with other applications. Because applications are standalone systems they can use the tools best suited for their purpose. A SOA also makes it easy to reason about scalability, since each application has a single purpose that must keep working exactly the same when system load increases.

Although the SOA provides several benefits it also introduces some complexities. Because of the distributed nature of the system it will be more difficult to ensure consistency. This means that most likely some sort of eventual consistency scheme will have to be used in certain situations. Additionally, it will be more difficult to execute transactional operations. These challenges will have to tackled on an application level, since the alternative monolithic application is not a valid alternative.

5.2 High level architecture

The platform consists of a number of *core applications/components*. These applications are essential for management and enforcement of the model as described in the previous chapter. The entry point for communication with the platform is a single component that makes use of these core applications. The goal of this entry point is protecting resources from unauthorized access. The next sections will discuss the goals of these core applications and how they fit together.

5.2.1 Core applications

The core applications map almost directly to the separate models as described in the previous chapter. This means that for every model there is a separate application which focuses on management of the model.

- **Node repository:** Provides an interface to interact with the graph model.
- **Application repository:** Provides an interface to interact with the application model.
- **Policy repository**: Provides an interface to interact with the policy specification model.
- **Permission repository:** Provides an interface to interact with the permission model.
- **Authentication:** Provides an interface to handle authentication related logic, which mainly entails converting authentication tokens to an authentication context and back.
- **IBM Connections:** The main data subjects in the system are employees. In order to interact with employees the platform uses the existing infrastructure of IBM Connections, a business social network platform. IBM Connections is integrated in the platform through a separate core application in order to have access to this data in the same manner as other data in the platform can be accessed.

These core components are similar to any other application that can be defined within the platform. However, without these core applications the platform can not function properly. Accompanying these core applications there are two client application for management purposes.

- **Application manager:** Provides a client interface that can be used by system administrators to define applications and their meta data, including policies.
- **Permission dashboard:** Provides a client interface that can be used by employees to manage permissions over their data.

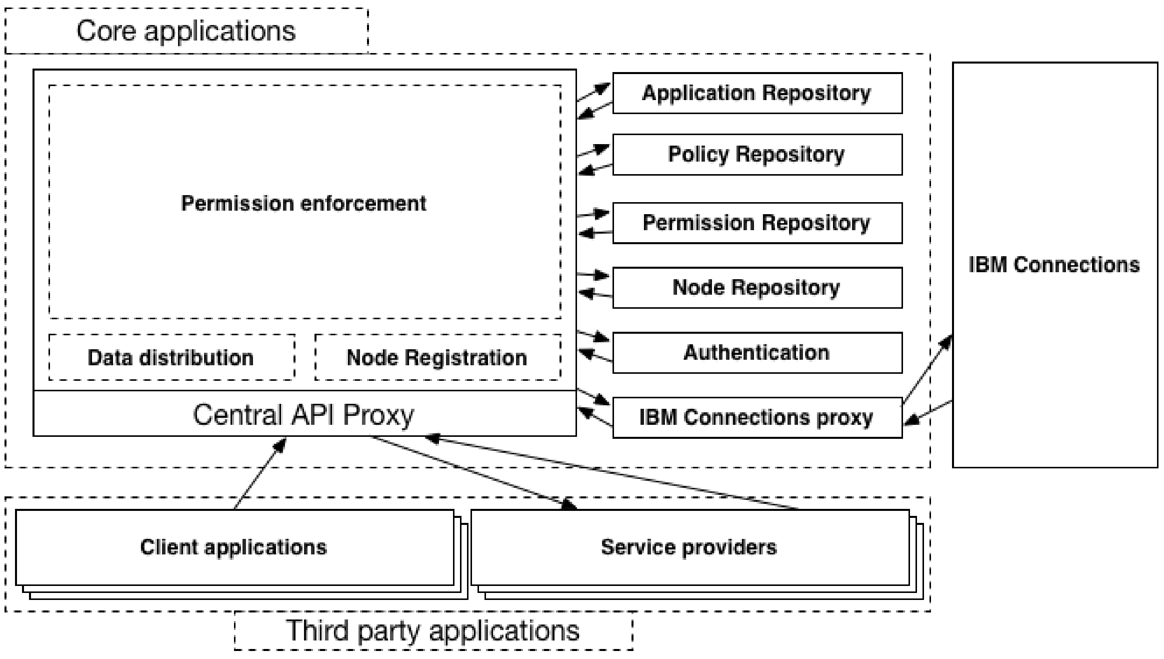


Figure 5.1: Platform architecture

5.2.2 Central API proxy

The entry point of the platform is a central API that processes API requests. In this central component requests are processed to determine whether access to certain resources should be granted. An overview of the architecture can be seen in Figure 5.1. This architecture shows resemblance to the XACML architecture as described in subsection 3.5.1. There are several other observations that can be made about this architecture.

Inter-application communication

First of all, every application within the platform only communicates directly with the central API proxy. Through this model, every request that is made within the platform can be controlled in order to ensure that access to resources is in compliance with user opt-ins.

Central API Proxy components

Within the central API proxy there are additional subcomponents, namely *permission enforcement*, *data distribution* and *node registration*. The main task of this platform is determining access rights to resources. This responsibility is handled by the permission enforcement subcomponent. The node registration subcomponent is used by applications to register nodes within the system. When a node is registered, its existence is documented in the node repository and default permissions are created in the permission repository. Finally, the data distribution subcomponent distributes data events to other applications.

5.2.3 IBM Connections integration

Interaction with IBM Connections is handled by the IBM Connections proxy application. The same principle can be used to create proxies to other services, such as for instance LinkedIn or Facebook. The use of this approach offers the benefit that access to third party services can be easily controlled throughout different applications in order to ensure that usage is in compliance with third party API policies and usage agreements.

5.2.4 Third party applications

Any application within the platform that is not a core application is considered to be a third party application. As can be seen in Figure 5.1 there are two types of third party applications, client applications and service providers. The difference between these applications is that service providers define additional services that can be invoked by other applications. Client applications on the other hand only interact with existing services.

5.3 Request processing

In order to elaborate on the goal of the core applications as given in the previous section the procedure for handling requests will be discussed. As previously mentioned the platform is accessible through a central API. Applications within the platform can make requests to this API, thereby getting the benefits of a privacy management layer. During these requests several decision points are used to determine the validity of a request. After verification of a request using the access control model requests are proxied to the targeted third party servers. After proxying additional meta data management is applied. The next sections will outline the various decision points used during API request processing.

For the proxying process some constraints have to be kept in mind. The privacy platform acts as an intermediary between actors making API requests to third party services. These API request will most likely be invoked through the use of client applications or by scheduled processes on servers. The downside with this approach is that the platform will introduce additional latency for these services. Performance of the platform is therefore important and latency should be as low as possible.

5.3.1 Authentication

As discussed in subsection 4.3.2 a request is made with a certain authentication context. In order to make requests possible a user of the platform first needs to authenticate. When a user has successfully authenticated through an external oAuth2.0 provider the API constructs a JWT as discussed in subsection 3.2.3. This construction is done by invoking the *Authentication* component. This component is in charge of authentication within the system. Specifically, this component exposes an API that allows for constructing new JWT tokens, but also for converting tokens to authentication contexts. After the JWT as been constructed it is passed to the application from which the login was requested. Using this token the application can now make requests on behalf of the user.

5.3.2 Token validation

After a token has been obtained an application can make an authenticated request to the API by including the token into each request using the HTTP authorization header. After a request is received by the central API the first step is determining the authentication context. There are two approaches for determining the authentication context.

The first approach requires the API server to make an HTTP request to the authentication component's token validation service. This however will negatively impact the overall latency of the platform. Since the authentication context is required before any access control decisions can be made, this first approach immediately introduces some latency.

Instead of invoking the authentication component another approach is to use the JWT standard for determining the validity of tokens directly. As previously mentioned each access token contains a signature. The authentication component uses an asymmetric signing algorithm to sign these access tokens. By using the authentication component's public key tokens can directly be validated by the API server. To allows this approach the authentication component exposes the public key through a service. This public key is cached by the API server. This approach allows for fast verification of access tokens and thus authentication contexts.

After the token is validated a final decisions can be made regarding the authentication context. When the token is invalid, the request is aborted. It is also possible to exclude a token in a request, in that case a default anonymous authentication context is used. An overview of the token validation process is given in Figure 5.2.

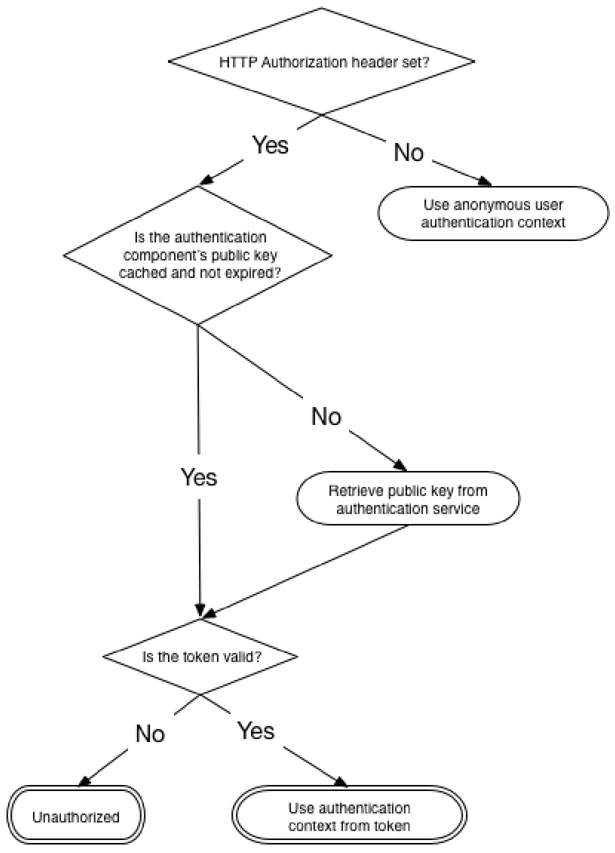

HTTP Authorization header set?
Yes
No
Use anonymous user authentication context
Is the authentication component's public key cached and not expired?
No
Yes
Retrieve public key from authentication service
Is the token valid?
No
Yes
Unauthorized
Use authentication context from token

Figure 5.2: Token validation

5.3.3 Application resolution

A request to the central API always targets a single service and a single service consumption. When a request is initially received all identifying features of the targeted service are available. Namely, the path of the request identifies both the target node type and the edge node type. From these properties the application offering the service can be deduced; When an edge node type is specified the service must be offered by the application that defined the edge node type, otherwise the service must be offered by the application that defined the target node type.

Using this information a query is made to the application repository to retrieve additional application meta data required for proxying the request. Namely, the API URL of the targeted service is needed for proxying requests. An additional performance improvement is introduced here to keep overall latency low. For this improvement the assumption is made that applications registered in the system are there for long term and that application meta data does not change too often. Using this assumption applications can be cached in memory for a short duration.

Assuming API URLs of around 100 bytes, this allows for storing a substantial number of API URLs in memory. This technique reduces the number of HTTP request needed for requests significantly. The application resolution process is outlined in Figure 5.3.

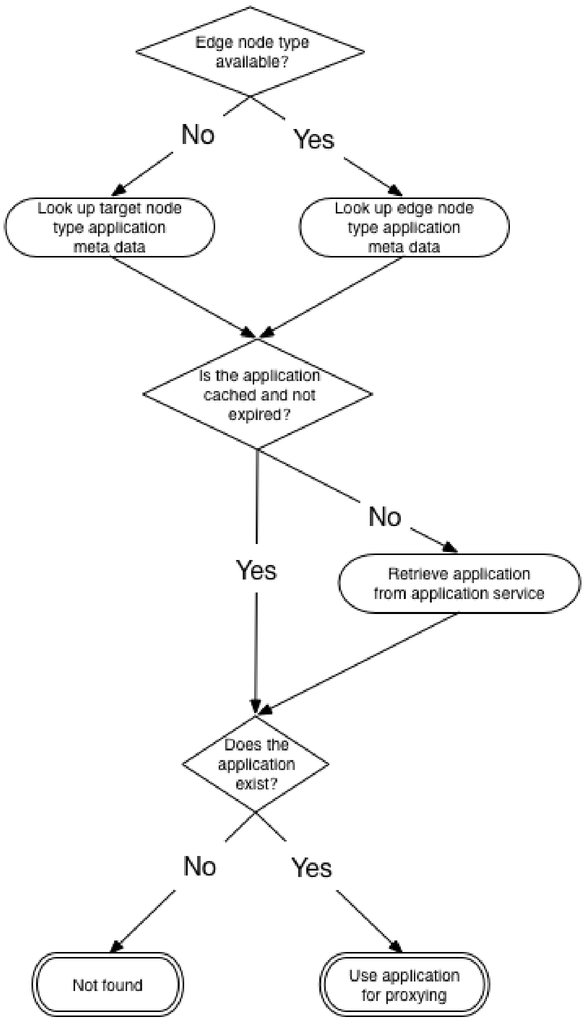

Edge node type available?
No
Yes
Look up target node type application meta data
Look up edge node type application meta data
Is the application cached and not expired?
No
Yes
Retrieve application from application service
Does the application exist?
No
Yes
Not found
Use application for proxying

Figure 5.3: Application resolution

5.3.4 Service resolution and service consumption resolution

As mentioned above, when a request is initially received all identifying features of the targeted service and service consumption are available. For services these features are the target node type, the edge node type and the service version. For service consumptions these features are the targeted service, the service consumption version and the service consumption identifier. The service consumption identifier is optional and if omitted a default identifier will be used. By using this approach no additional lookups are needed regarding service and service consumption details. Although this provides low latency overhead it should be noted that at this point in request processing there is no verification of whether the service and service consumption actually exist. This verification is done implicitly through permission look ups in further steps.

5.3.5 Permission lookups

After the service and service consumption being targeted are determined the next step is to determine whether there is an applicable permission available. There are two permissions that could potentially be available; a service permission and a service consumption permission. Using the information from the previous steps these permissions can be uniquely identified; When the request targets a specific node, the *target node*, the permission must be provided by this target node. Otherwise, permission of the application providing the service is needed. Using this information two queries can be made to the permission component to retrieve permission meta data. If no permissions are found the request is aborted, otherwise the next step is executed. An overview of this step is given in Figure 5.4.

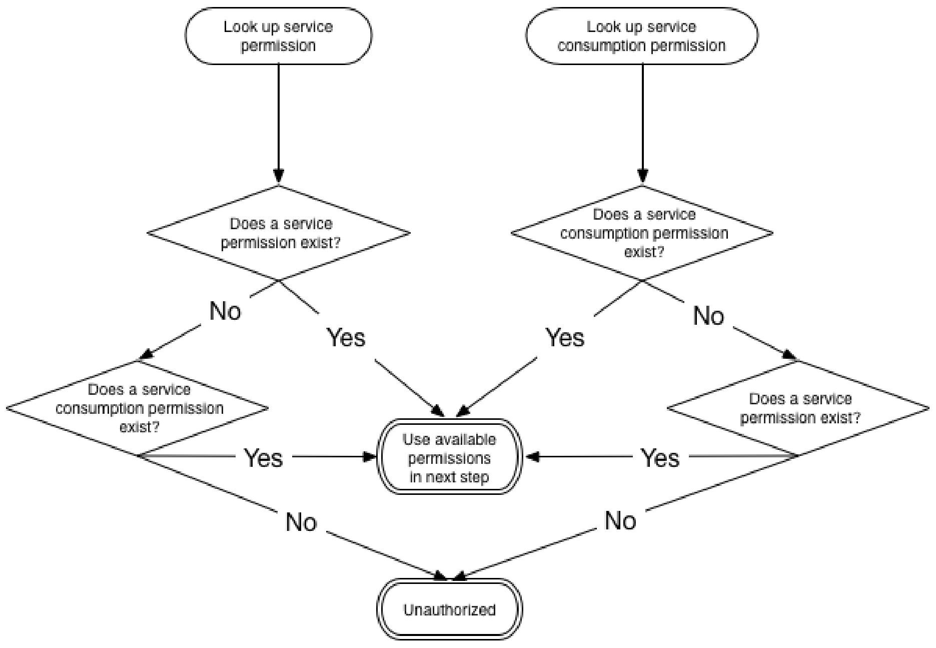

Look up service permission
Look up service consumption permission
Does a service permission exist?
Does a service consumption permission exist?
No
Yes
Yes
No
Does a service consumption permission exist?
Does a service permission exist?
Yes
Use available permissions in next step
Yes
No
No
Unauthorized

Figure 5.4: Permission look ups

5.3.6 Permission evaluation

After possible permissions have been found they must be evaluated. As was discussed in section 4.3 a permission has a list of permission rules. Each of these rules states a relation with some arbitrary node. A permission is accepted if the node from the authentication context satisfies the relation to the node from any permission rule. The list of permission rules together with the node from the authentication context can thus be transformed into a list of relations. This list of relations is send in bulk to the nodes component, which then evaluates whether any of these relations exists. If the nodes component successfully identifies an existing relation the request is accepted. When both the service permission and service consumption permission are evaluated successfully the service consumption permission takes priority in order to keep access logs as specific as possible. Since there are potentially a lot of permissions caching is not yet used for this component. An overview of this process is given in Figure 5.5.

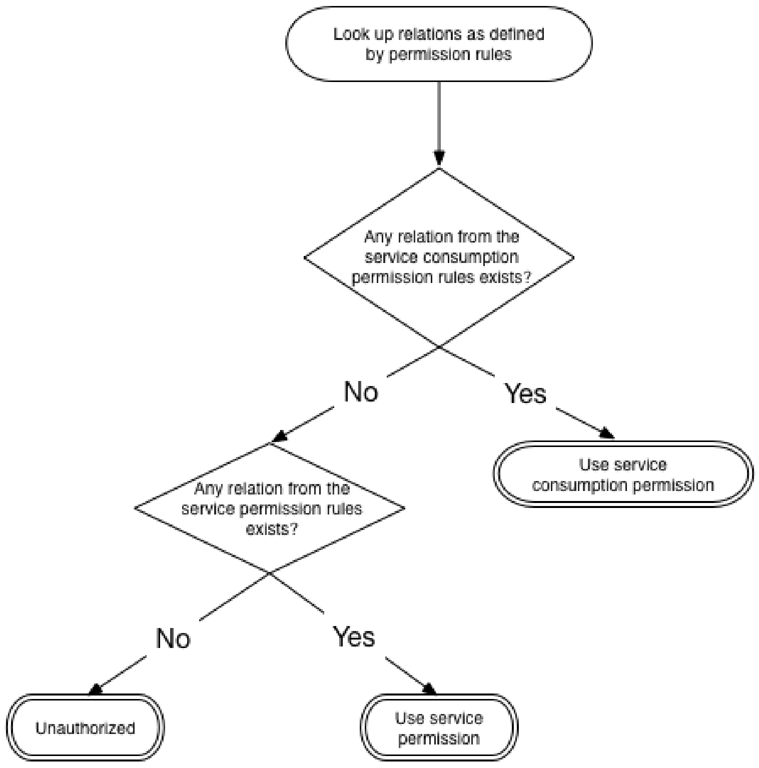

Look up relations as defined by permission rules
Any relation from the service consumption permission rules exists?
No
Yes
Use service consumption permission
Any relation from the service permission rules exists?
No
Yes
Unauthorized
Use service permission

Figure 5.5: Permission evaluation

5.3.7 Request proxying

After determining that a request is valid the central API proxies the request to the application that has defined the service in question. The proxy request can be streamed from the application offering the service to the client making the request to reduce the memory footprint on the central API. Note that the central API is not concerned with the actual contents of the proxy request, thus applications are free to choose how they represent data.

5.3.8 Node administration

An additional step in the request processing pipeline is the administration of nodes. In this step, HTTP headers of the resulting proxied request are inspected to determine whether any changes to the underlying graph model have occurred. By looking at the request HTTP method, the resulting HTTP status code and additional response headers the exact changes to the graph model can be determined.

An additional observation here is that this system can be used to introduce an eventual consistency model. Such a model is needed due to the fact that during request proxying there are several things that can go wrong. When a request is partially executed it is possible that the graph model as stored in the node repository is not consistent with the graph model as stored by a particular application. In order to gradually improve the consistency of the node repository future responses can be used to determine undocumented changes to the graph model. For instance, when an application responds with the *Not Found* status code on a request targeting a specific node, this means that the targeted node does not exist anymore. This information can thus be included in the node repository.

HTTP Method	**HTTP Status**	**Additional headers**	**Action**
POST	Created (201)	Location, request path	Create new node, trigger distribution event
PUT	Accepted (200)	Request path	Trigger distribution event
DELETE	Accepted (200), No Content (204)	Request path	Delete node
GET	Not Found (404)	Request path	Delete node if it still existed
GET	Accepted (200)	Request path, ownership meta data	Create node if it did not exist
PUT	Accepted (200)	Request path, ownership meta data	Create node if it did not exist

5.4 Implementation

The core components all have a similar setup. Each component exposes an API according to the application model as given in section 4.2. This means that each component is deployed as a separate web application. Because of the shared structure of core components a similar technology stack is used for all components.

5.4.1 Java

The core components are all written using the Java programming language [1]. The choice for this language, although partially subjective, is that it provides a solid basis for building enterprise applications. There are various useful standards and high quality libraries that can be used during implementation to construct an efficient and scalable platform.

5.4.2 Platform: IBM WebSphere Liberty

Considering this project is conducted at IBM there are some restrictions on third party technology usage. In order to reduce the need for external dependencies the implementation makes use of IBM WebSphere Liberty[2]. WebSphere Liberty is a lightweight Java EE application server providing direct access to open source implementations of several standards.

5.4.3 Project structure

The main structure for each project is as follows. First of all, each core component generally defines some node types. Thus, each core component also has a single module containing classes describing node types. Secondly, each application implements services targeting these node types in a separate module. The implementation of these services is itself structured in three main sub modules. First of all there are several interfaces that describe the REST services using JAX-RS annotations. Secondly there is a submodule that implements these interfaces using a specific persistence technology. Finally, there is an application module that combines these implementations into a single servlet. The result of this project structure is illustrated in Figure 5.6.

Whereas the node type modules only describe classes without any additional logic attached to it, the service implementation bundles contain actual component specific logic. Thus, as a general rule the only interdependencies between core components are that of components using node type definition bundles of other components. Other component related logic should be hidden from other components. This way, each component knows about data stored in other components and how this data can be manipulated through services specified in the application model, but the actual logic of how this manipulation works is hidden.

[1] `https://docs.oracle.com/javase/specs/jls/se8/html/index.html`

[2] `https://developer.ibm.com/wasdev/websphere-liberty/`

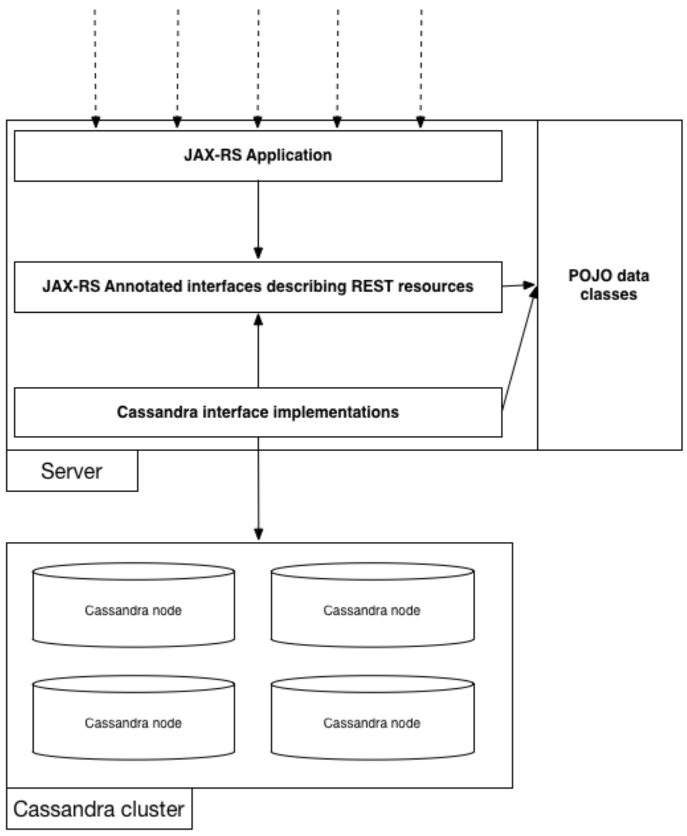


Figure 5.6: Core component architecture

5.4.4 Persistent storage strategy

An important part of the platform is the persistence storage of the various models. An important consideration when choosing a persistent storage strategy for the core components is the way in which the components will query data. The main choice between technologies is the choice between SQL or NoSQL. SQL products tend to expose a more complicated model, whereas NoSQL solutions provide a less extensive model but instead provide more focus on other features such as horizontal scalability. In an enterprise environment scalability is a major concern. The way in which data is queried within the system is also more focused on determining existence rather than providing some relational result sets. Functionalities such as complicated joins are therefore less important. For this reason, NoSQL is used for core components.

There are several different types of NoSQL data models. A brief overview is given in [8]:

- *Key-value stores* are systems that simply provide a scalable way to a bind keys to values.

A common use case for such models is caching. Examples of key-value stores are Redis[3], Memcached[4] or Oracle NoSQL Database[5].

- *Document stores* provide the functionality to store document based models. The term document is very broad but common document stores offer functionality to store data in some structured form such as XML or JSON. Example document stores are Apache CouchDB[6] or MongoDB[7].
- *Extensible Record Stores* are systems that use a model in which regular database tables exist, but rather than providing relations between multiple tables such as is done in SQL through joins, these models rather focus on simple row based access only. Additionally, rows can grow horizontally over time, meaning that new columns can be introduced. Extensible record stores can be seen as a form of a key value store. Examples of such systems are Google BigTable[9], Apache HBase[8] or Apache Cassandra[9].
- *Linearly scaled relational databases* are standard SQL databases that provide some way in which linear clustering can be achieved. For such systems there can be two approaches in realizing this; the existing database can be adjusted to directly offer clustering functionality or an additional system can be built on top of existing SQL solutions that handles clustering by itself. An example of such a use of traditional SQL database is for MySQL Cluster[10].
- Finally, other models include *Graph databases* or *Object Orientated databases*. Graph database provide scalable ways to analyze graphs while object orientated databases focus on storing and querying models in an object orientated format.

When choosing a technology for persistent storage in the core platform components some additional features have to be taken into account. Each of these models provides advantages in different situations and each one differs in terms of functionalities such as fault tolerance. As mentioned earlier an important part of the platform is determining existence of data, mainly permissions and relations between nodes. For this reason, efficient lookups are desired. The data stored is also rather simple in terms of structure. Thus, as a basis for implementation the extensible record store model is chosen. The main argument for this storage model is that plain key-value stores are rather limited in terms of functionality and document stores are more focused towards storage and manipulation of larger documents rather than small updates to existing documents. Linearly scaled relation databases do not provide any additional required functionality over extensible record stores. Finally, graph databases are also less applicable to the existence queries that will be used.

[3] `http://redis.io/`
[4] `https://memcached.org/`
[5] `http://www.oracle.com/us/products/database/nosql/overview/index.html`
[6] `http://couchdb.apache.org/`
[7] `https://www.mongodb.org/`
[8] `https://hbase.apache.org/`
[9] `http://cassandra.apache.org/`
[10] `https://www.mysql.com/products/cluster/`

The remaining choice is between technologies. Although arguments can be made for different extensible record stores the choice has been made to use Apache Cassandra. Apache Cassandra provides useful functionality such as bloom filters[34] for efficient existence checking, or functionality such as secondary indexing. The choice for this technology partly remains a matter of preference. For future extensions more specialized storage models for different core components could be more useful, such as for instance a distributed graph database for storing the node graph. Currently however there is no need for complex analysis or querying of the node graph and thus use of such a system is avoided.

5.4.5 Node model implementations

In order to describe nodes within core components a convention has been developed that allows for describing the graph model within code. This convention is enforced using annotations and custom annotations parsers. By using these technologies there is a consistent way of describing node type classes. Implementation of these classes is specific to the Cassandra storage technology. The main reason behind this is that this way Cassandra annotations can be utilized to define storage requirements. Nevertheless the main structure of this node type description convention can be reused for other storage engines.

As an example, consider the *RelationType* node type. This node type describes relation types within the graph model. Note that the graph model applies on components of the model itself. The code below shows the definition of this node type. Some code has been omitted for brevity.

```
@Table(
    keyspace = RelationType.NODE_TYPE_APPLICATION_ID,
    name = RelationType.NODE_TYPE_NAME)
@NodeModel
public class RelationType extends AbstractCassandraModel {

  @NodeTypeApplicationId
  public static final String NODE_TYPE_APPLICATION_ID =
      ApplicationsApplicationDescription.APPLICATION_ID;

  @NodeTypeName
  public static final String NODE_TYPE_NAME = "relation_type";

  @NodeTypeId
  public static final String NODE_TYPE_ID =
      NODE_TYPE_APPLICATION_ID + $ + NODE_TYPE_NAME;
}
```

The code above shows several annotations. There are two kinds of annotations; Cassandra specific annotations (Datastax Object-mapping API [11]) and graph model annotations. The first

[11] https://docs.datastax.com/en/latest-java-driver/java-driver/reference/objectMappingApi.html

set of annotations describe how the object is persisted in Cassandra. The second set of annotations describe the graph model in code. In this example the following annotations are of interest; **@NodeModel**, **@NodeTypeApplicationId**, **@NodeTypeName**, **@NodeTypeId**. These annotations describe that the particular class is a model describing a particular node type within the platform. Each NodeModel has an application ID and a name. The combination of the two uniquely describes a node type within the platform. The values for these properties are constant and the graph model annotations are used to define them in a conventional way.

Custom annotation parsers ensure that each class annotated with the **@NodeModel** annotation always define the fields NODE_TYPE_APPLICATION_ID, NODE_TYPE_NAME, NODE_TYPE_ID and TARGET_NODE_PATH. This way, service description interfaces can reference node types through constants, making misspelling bugs less likely. An example of usage of these constants can be seen in the REST interface code shown in 5.4.7:

```
@POST
@Path(User.TARGET_NODE_PATH + "/" + Application.NODE_TYPE_ID)
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
@JsonView(Views.Private.class)
Application create(
    @PathParam(ID) String userId,
    Application application) throws StorageException;
```

5.4.6 Indexing

An important part of the storage implementation is that efficient lookups should be possible. This requirement is realized by making use of Cassandra functionalities. In Cassandra each model defines a primary key. This primary key is used to determine where data should located in the Cassandra cluster. Queries upon this primary key can efficiently be executed. Primary keys can also consist of multiple fields.

This functionality is especially useful during proxy request handling since it allows for efficiently looking up desired meta data. When a proxy request is made there is a certain amount of information available about the service being targeted; namely the target node type, the edge node type, the HTTP method and the service version. This information is sufficient for determining the exact primary key of the targeted service.

This indexing behavior is materialized in another convention for implementation of NodeType classes; Each NodeType class defines some meta data about how they are uniquely identifiable. The first type of information is a Regex pattern[15]. This pattern describes how the model is uniquely defined in the form of a string. Secondly, annotations are used to describe which fields are part of this representation and in what order. Using this convention several functionalities can be created that allow for retrieving the unique identifier for a NodeType class instance and for converting a unique identifier back to a NodeType class instance.

Consider the following class definitions describing some NodeTypes within the platform (details have been omitted for brevity):

```
@NodeModel
public class Application extends AbstractModel {

  @NodeIdPattern
  public static final String NODE_ID_PATTERN =
      "(" + NAME_PATTERN + ")";

  @IdSegment(0)
  private String name;
}
```

```
@NodeModel
public class NodeType extends AbstractModel {
  @NodeIdPattern
  public static final String NODE_ID_PATTERN =
        "(" + Application.NODE_ID_PATTERN + ")"
      + $
      + "(" + NAME_PATTERN + ")";

  @IdSegment(0)
  private String applicationId;

  @IdSegment(1)
  private String name;
}
```

In this code, NAME_PATTERN is a constant that has the pattern $[a-zA-Z][a-zA-Z0-9\_]*$ and $ is a constant that defines a default ID separation character, which in this case is a dot.

The first class definition describes the Application node type. This code shows that an application is uniquely identifiable directly by its name. Subsequently, the NODE_ID_PATTERN consists of a single Regex capture group for a name and there is one field, the name field, which is part of the node id pattern.

The second class definition describes the NodeType meta node type. As can be seen from this code a NodeType instance can be uniquely identified by an application id and a name. The NODE_ID_PATTERN consists of two capture groups, one for the application id and one for the name. The order of the fields in this NODE_ID_PATTERN is indicated by the **@IdSegment** annotation.

The next code segments showcase how these conventions can be used throughout the implementation. These examples use the NodeType class to describe the gamification application and a hypothetical images application. Using the id conventions the following behavior is possible through generic methods defined in *AbstractModel*:

```
Application gamificationApplication = new Application();
gamificationApplication.setName("gamification");

gamificationApplication.getId(); // "gamification"

Application parsedApplication = Application.parse("images");
parsedApplication.getName(); // "images"


NodeType achievementNodeType = new NodeType();
achievementNodeType.setApplicationId(
  gamificationApplication.getId());
achievementNodeType.setName("achievement");

achievementNodeType.getId(); // "gamification.achievement"

NodeType parsedNodeType = NodeType.parse("images.image");

parsedNodeType.getApplicationId(); // "images"
parsedNodeType.getName(); // "image"
```

This mapping from strings to fields provides the opportunity to directly perform Cassandra queries using models with multiple primary keys. This convention thus is mainly useful in the context of Cassandra interaction. The following example shows how this convention can be used to look up meta data about services:

```
Service serviceSignature = new Service();

serviceSignature.setTargetNodeId(request.getTargetNodeTypeId());
serviceSignature.setEdgeNodeTypeId(request.getEdgeNodeTypeId());
serviceSignature.setHttpMethod(request.getHttpMethod());
serviceSignature.setVersion(request.getVersion());

ApiClient api = new ApiClient();

// HTTP request to Application repository API
// returns the service together with all its meta data
api.get("applications.service/" + serviceSignature.getId());
// Equivalent
api.get(Service.NODE_TYPE_ID + "/" + serviceSignature.getId());
api.get(serviceSignature);
```

5.4.7 JAX-RS 2.0

For constructing HTTP interfaces the JAX-RS 2.0 [27] standard is used. This standard provides a simple way to expose services over HTTP. There are several implementations available for this standard. The choice of this standard is driven by the fact that this standard is commonly used when it comes to building HTTP APIs. Alternatives offer almost identical functionality but are not as widely supported as JAX-RS.

JAX-RS uses annotations to describe how services can be accessed through HTTP. In the platform implementation this is utilized as follows; Each core component specifies a number of interfaces that determine how the data model can be interacted with. These interfaces are annotated with JAX-RS annotations. An example of such an annotated interface is given below:

```
@Path("/")
public interface ApplicationsResource extends Resource {
  @POST
  @Path(User.TARGET_NODE_PATH  + "/" + Application.NODE_TYPE_ID)
  @Consumes(MediaType.APPLICATION_JSON)
  @Produces(MediaType.APPLICATION_JSON)
  @JsonView(Views.Private.class)
  void create(
    @PathParam(ID) String userId,
    Application application,
    @Suspended AsyncResponse asyncResponse);

  @GET
  @Path(Application.TARGET_NODE_PATH)
  @Produces(MediaType.APPLICATION_JSON)
  @JsonView(Views.Private.class)
  void get(
    @PathParam(ID) String id,
    @Suspended AsyncResponse asyncResponse);

  @DELETE
  @Path(Application.TARGET_NODE_PATH)
  void delete(
    @PathParam(ID) String id,
    @Suspended AsyncResponse asyncResponse);

  @GET
  @Path(User.TARGET_NODE_PATH + "/" + Application.NODE_TYPE_ID)
  @Produces(MediaType.APPLICATION_JSON)
  @JsonView(Views.Private.class)
  void getAllByUserId(
    @PathParam(ID) String userId,
        @Suspended AsyncResponse asyncResponse);

    @GET
    @Path(Application.NODE_TYPE_ID)
    @Produces(MediaType.APPLICATION_JSON)
    @JsonView(Views.Public.class)
    void getAll(
            @Suspended AsyncResponse asyncResponse);
}
```

This Java code snippet shows how JAX-RS annotations are used to describe access to applications in the Application data model. This interfaces shows some additional usage of functionality provided by JAX-RS:

1. The service descriptions work with plain Java objects. When requests are made against this interface these methods are called directly. However, transportation is done using the JSON format. In order to make this convention possible a JSON serializer is used as middle-ware. This middle-ware converts JSON input to plain Java objects and back. For the implementation Jackson [12] has been used, but several alternative solutions exist.

2. Exceptions are not handled on a service level. Rather, middle-ware is used that handles service invocation exceptions and provides descriptive JSON results to API users.

3. Every service description accepts an additional argument, an *AsyncResponse*. By using the **@Suspended** annotation JAX-RS provides the ability to respond to request asynchronously.

Behind the annotated interfaces describing the REST capabilities there are implementations using a particular storage technology. Using this pattern implementations can be developed using a different storage technologies. The access interface is known and by subsequently implementing exactly this interface a new storage technology can be utilized. As an initial implementation Cassandra was used for each of the core components but future work might focus on using more suitable tools for the different core components.

5.4.8 Asynchronous programming stack

As mentioned in the previous chapter the JAX-RS services make use of asynchronous features of the JAX-RS standard, namely the AsyncResponse. This feature is combined with various other libraries and standards, resulting in a event driven implementation.

First of all the Cassandra queries are executed using the Datastax Java Cassandra driver [13]. This driver provides an asynchronous API, for which connection pooling is utilized. This API works with *ListenableFutures* from the Google Guava Java library[14]. ListenableFutures, as the

[12] `https://github.com/FasterXML/jackson`
[13] `https://github.com/datastax/java-driver`
[14] `https://github.com/google/guava`

name suggests, represents results that are available sometime in the future. Subsequently listeners can be attached to this future that will be notified when the result is available. Within core components ListenableFuture instances are converted to Java 8 *CompletableFuture* instances. CompletableFutures represent a similar abstraction as ListenableFutures, but CompletableFutures provide a convenient Java 8 styled API.

Another point at which asynchronous tools are utilized is during inter component requests. The central API uses services provided by core components during request proxying. Thus, to make these requests possible without introducing too much overhead several requests have to be executed in parallel. This is implemented using the AsyncHttpClient2 library [15]. This library can be backed by Netty[16], an event driven network application framework. This library also provides access to CompletableFutures.

Finally, the in memory caching implemented in the central API makes use of the Caffeine caching library[17]. This library provides an API that works with CompletableFutures and allows for easily specifying caching details such as expiration requirements and asynchronous value loading. Due to the use of the CompletableFuture API this library can easily be integrated with the rest of the asynchronous technology stack.

5.4.9 Implementation of proxy request handling

As can be derived from the proxy request handling procedure as discussed in section 5.3 there are a number of lookups that take place during proxy request handling. In order to implement this process efficiently an asynchronous non blocking implementation is used. By using the CompletableFuture API provided by Java 8 asynchronous events can be chained concisely.

When request are made to the central API the response is deferred using the JAX-RS @**Suspended** response annotation. By using this annotation the response can be put on hold until enough data is available. This is in contrary to a JAX-RS method without a suspended response in which the executing thread will block until processing is completed. The required internal calls to core components are wrapped in CompletableFutures. These futures are chained in such a way that the result is a single CompletableFuture instance that describes the proxy request. When this CompletableFuture is completed the suspended response is resumed by proxying the request to the application defining the targeted service. When in any stage of the internal system calls an error occurs this will be propagated to the resulting CompletableFuture, which in turn will resume the suspended response with the appropriate error message.

Proxy request steps

There are several lookups that have to be made before a proxy request is executed. These lookups ensure that access to data is allowed. As mentioned above these lookups are done in parallel to keep latency low. However, some lookups are dependent on other lookups. Thus, there are several required sequential steps needed during the proxying process. The order of these steps is outlined in Figure 5.7.

[15] `https://github.com/AsyncHttpClient/async-http-client`

[16] `https://github.com/netty/netty`

[17] `https://github.com/ben-manes/caffeine`

When a request is received the first step is determining the application meta data and authentication context. Additionally, since all information required to identify the targeted service together with the targeted node is contained in the request the service permission can be retrieved directly. The next step is looking up the service consumption permission and evaluating the service permission. In order to identify the targeted service consumption the authentication context is needed because this context contains information about the application making the request. For the service permission the authentication context is needed because this context contains the user making the request. Finally, when the service consumption permission is obtained it can be evaluated. When either the service permission or the service consumption permission has been successfully evaluated the proxy request can continue.

After the headers are obtained during the proxy request additional node administration processes can be started. This administration is executed in isolation of the proxy request. The rest of the proxy response body is streamed to the client.

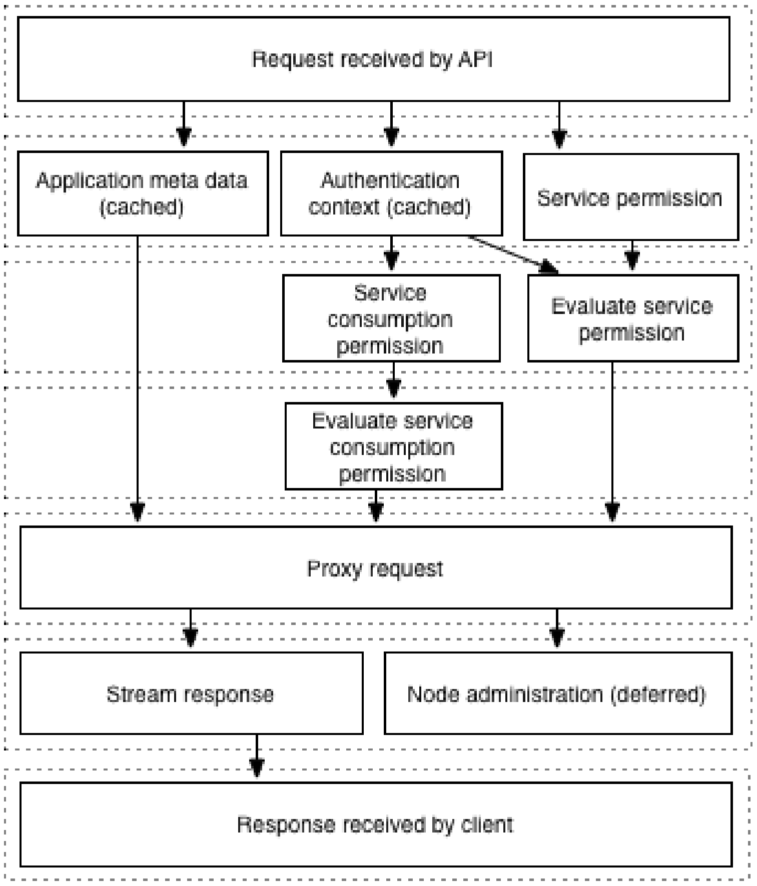

Request received by API
Application meta data (cached)
Authentication context (cached)
Service permission
Service consumption permission
Evaluate service permission
Evaluate service consumption permission
Proxy request
Stream response
Node administration (deferred)
Response received by client

Figure 5.7: Proxy request steps

5.5 Third party application responsibilities

In order for third party applications to make use of the functionality of the platform they need to perform some additional administration. In return, the application receives a standardized manner to handle privacy and policies. Administration includes defining application services and policies. After services have been defined they can be requested through the central API.

When a node is created through a service the application is responsible for registering this node within the platform. An application can only register nodes of a node type it defined itself. Similarly node deletion has to be registered in order for the system to remain consistent. An additional advantage of node registrations is that these events can be propagated to other applications. This way, a real-time flow of events can be established between applications.

As outlined in subsection 5.3.8 node administration within the system is done at proxy time. This means that when a proxy request is executed, the result of this proxy request is used by the central API to determine changes to the graph model. For this approach to work correctly applications must correctly specify the appropriate HTTP headers.

Besides the basic ownership administration that is automatically applied with the help of contextual information applications can also provide additional relations that might be of use when resolving permissions. For instance, a particular third party application can introduce a friendship relation between nodes. This friendship relation can be useful for end users to incorporate into the permission process; One user can configure that their profile is visible to friends only. In order for this system to work applications are allowed to create additional relations between nodes. Some restrictions that apply to this process are:

1. Applications are considered owners of node entries in the Nodes repository
2. Applications can only own nodes with a node type that they have defined
3. Applications can only create new relations where the target node is a node that they own

5.6 System consistency

As discussed in subsection 5.3.8 having applications register status of nodes within the system does introduce some downsides, the main one being additional complexity in keeping the platform consistent. When an application creates a node there is always a chance that creation of the node within the application succeeds but the registration fails. Failure can happen for any reason, such as because of a programming error but also because of other events such as hardware failure. This means that eventually a situation can occur in which the platform and the application are in disagreement over the status of some node. In order to keep the system consistent the eventual consistency strategy is applied. Additionally periodic consistency checking can be planned between the central API and applications.

5.7 Reflection on scalability

An important requirement of the system is scalability. Latency overhead should be minimal, even when multiple applications expose high traffic services. For these applications the system acts as a reverse proxy and while this has advantages for the implementation, it does make the system a central bottle neck. Therefore, it is essential that the system can easily be scaled.

The service orientated architecture provides a basis for scalability. In this architecture, each individual component can be scaled and optimized separately depending on how much traffic it receives. In order to achieve scalability for core components, each component is constructed as a fully stateless application. This means that each component can theoretically be replicated and put behind a load balancer. Since each core repository makes use of Cassandra for storage the persistence layer can also be scaled out when the need arises.

Since the central API will be the biggest bottle neck in the architecture the importance of scalability is especially applicable to this part of the system. For this central API the same stateless approach was used. This means that the central API applications can be replicated when needed to allow for higher overall throughput.

For authentication the question of scalability is slightly more complicated. Because of the use of the JWT standard the authentication server needs secret keys for token signing. When servers are replicated these secret keys need to be taken into account. However, this is a common challenge and several solutions exist, such as for instance Hardware Security Modules.

5.7.1 Performance limitations

Due to the required sequential steps in the proxying process there is a substantial overhead that is added to API request that is not present when requests are made directly towards underlying services. Thus, applications making use of the platform must be aware of the performance implications paired with using the platform. Since the first layer of request can mostly be cached, there are two additional internal round trips required before requests can be executed. In the current implementation communication between components uses HTTP directly. Although this provides some conveniences in terms of implementation, it does provide a lot of unnecessary overhead. When HTTP is used then each request requires several handshakes for setting up a connection. These connections are not persisted between request. This is obviously not desirable, especially for services that are invoked often such as the permission services and the node services. For this a possible solution is the use of persistent HTTP connections. Alternatively these communications can utilize other communication technologies, preferably with persisting connections.

Chapter 6

Design validation and reflection

As part of the validation of this work discussions have been conducted with IBM employees from different departments. In these discussions different perspectives have been taken into account to validate whether the model is indeed a solution to the problem. First the high level goals and results of this platform have been discussed with development management. Secondly technical implementation details have been discussed with a software consultant.

6.1 High level goals and results

In order to validate the resulting system discussions have been performed with different employees of IBM. During the discussions of high level goals and design of the platform numerous questions have come up. In the next sections these questions will be outlined together with clarifications and reflections.

6.1.1 Who is responsible for management of application service descriptions?

A concern with a centrally managed platform such as presented in this work is that it requires dedicated efforts to maintain application meta data. These efforts require additional managerial decisions such as who is responsible for maintaining this meta data.

An argument against this concern is that it is not strictly bound to a centralized platform and thus to this solution. In any case, when applications deal with sensitive personal information there is a need for transparency towards end users. The question of responsibility would remain in a environment in which data sharing is handled in a non centralized way. When this meta data is defined in a single location it can become easier for end users to inspect this meta data which would in turn improve transparency. Therefore, a centralized repository of such meta data can actually be a beneficial addition to this concern.

Concluding, the question of responsibility is one that can come up in multiple different approaches to the challenges faced in this work. This work states that it would be better to formalize policies over responsibility rather than letting application developers handle this by themselves. This solution would be viable in both a centralized as well as a non-centralized platform, but we argue that in a centralized approach dealing with responsibility could be more formalized and concrete than in a non-centralized approach.

Concretely this could lead to a situation in which application meta data is created by application developers and approved by other departments within an organization. This way, more control can be applied to data usage to ensure that every application respects company policies with regards to data usage as well as third party data usage. By incorporating different expertise into the process of application meta data approval the quality of the overall meta data can also be increased. To elaborate, the technical details of application meta data are better suited for application developers while legal aspects are better suited for people with expertise in this area.

6.1.2 Why does the system act as a proxy between servers instead of direct interactions between service consumers and service providers?

A goal of the platform as implemented in this work is making data sharing between applications easier while still respecting privacy. When direct interactions are allowed between service consumers and service providers this means that there are several additional tasks that the service provider has to execute. Mainly, these service providers would have to incorporate some authorization and authentication logic into their functionality. This would introduce a large overhead for every application and thus increase the complexity for sharing data. For this reason, a more fitting approach is to incorporate this logic into a single point, thereby reducing the needs for duplicate implementation of this logic across various applications. By using a single point at which these sensitive access control decisions are made it will also be easier for application developers to incorporate a consistent and secure implementation of this functionality into their applications.

6.1.3 How are policies enforced throughout various applications?

As noted in subsection 2.5.7 the design of the platform has focused on a situation in which multiple development teams within an organization develop applications that aim to benefit employees in the context of an Inclusive Enterprise. Within this context, it is assumed that these application developers act in good faith and thus a part of the responsibility of policy enforcement lies with the application developers themselves. Nevertheless, it could still be possible that some developers misinterpret policies in some way or that some developers purposefully misuse their data access privileges. To combat these situations a formal approval process could be introduced that imposes some quality requirements and regulations on applications interacting with the Inclusive Enterprise platform. Additionally, as also stated in subsection 2.5.7, a possible future addition to the system is a reporting utility that can be used to bring to light malicious use of the platform.

6.1.4 How can this platform efficiently be utilized by development teams?

In order for this platform to be efficiently usable by development teams there needs to be an understanding among these teams about how such a system could benefit their applications. A key concern here is that adaptability could be low due to current practices already fulfilling some needs from an application developer's perspective. However, in most of these situations the user side of the problem is not considered. Therefore, additional requirements regarding use of personal data should be highlighted together with the ways in which this platform aims to address them. By clearly explaining the goals of this initiative there should be a proper motivation to incorporate the platform into applications.

When the motivation to use this system is indeed present the next step is to provide proper documentation and examples of platform usage. For these resources this thesis report can be used, as well as platform code documentation. For example applications making use of the platform the core components themselves can be inspected, as well as the management client applications developed as part of this work.

6.1.5 Is such a fine grained access control model maintainable for end users?

The model constructed in this work offers a very fine grained access control model. The danger of such a model is that it can become unusable by end users of the system. This is due to the fact that there are so many options to configure that overall configuration can become very confusing. This can especially become a problem when versions are upgraded frequently within the system. Such a situation would require end users to also frequently reevaluate policies.

The argument against this concern is that the model itself already provides ways in which granularity can be reduced. Mainly this involves the use of subset services, in which all nodes of a certain node type belonging to a certain target node are returned. Using these services, employees can grant applications access to numerous data nodes using only a single permission.

Secondly, a solution to tackle this potential problem is by means of innovative GUI designs. Such designs can reduce the cognitive load required for the underlying model by providing some layer of abstraction. For instance, such an interface can provide convenient methods to configure permissions for multiple nodes at once. Additionally, such a GUI can provide easy to process overviews of policy updates, showing only updated parts of a policy. Such a design falls outside the scope of this work, but provides an interesting opportunity for future work.

6.2 Discussion of the implementation

Besides the high level goals the implementation has been discussed with a software consultant within IBM. During this discussion some questions were raised, which will be outlined in the upcoming sections.

6.2.1 OSGi

An initial version of the implementation made use of OSGi [1]. OSGi is described as a dynamic module system for Java. This technology provides the basis for highly modular Java applications. WebSphere Liberty also provides support for deployment of OSGi applications. A big advantage of OSGi is that development within this system is pushed towards a modular design.

[1] `https://www.osgi.org/`

Additionally, OSGi provides opportunities with regards to deployment. Through OSGi sub-components of a system can be updated while other parts of the system remain running. This functionality could benefit update cycles by providing minimal downtime. Another advantage could be the availability of distributed OSGi implementations such as Apache CFX Distributed OSGi[2]. By utilizing these standards existing tooling can be used for implementing a scalable system. In distributed OSGi modules can be exposed over networks, thereby providing a high level of abstraction over distribution. Under the hood Apache CFG can communicate using SOAP (Simple Object Access Protocol) over HTTP or even JAX-RS.

The main question for this point is what the reasoning is for not using these technologies and instead implementing a REST architecture directly using JAX-RS. There are several reasons for this decision, which will be discussed in the next sections.

Integration with OSGi tools

First, the Eclipse tooling provided by WebSphere Liberty does not integrate fluently with other OSGi tools such as Bnd[3] and Bndtools[4]. These tools provide an abundance of functionality that helps with the development of OSGi bundles. For instance, Bnd can take care of a lot of the administration involved in defining OSGi bundle manifests. Combined with Bndtools this allows for fast development and fluent dependency management. Unfortunately these tools do not work in conjunction with the IBM WebSphere Liberty Eclipse tools[5]. In contrast, these tools provide a lot of functionality in terms of development to WebSphere Liberty, especially with automatic deployment to a local development server. This allows for quick development cycles with direct feedback. However bundle manifest administrations are less automated and thus error prone, meaning that dependency management leaves a lot to be desired. The same goes for the testing environment. Concluding, the integration issue meant that a choice had to be made between several functionalities, all of which were very useful. Unfortunately no solution existed that provided the best of both worlds. Finally, the WebSphere Liberty OSGi environment caused some issues with several third party libraries.

Expected deployment practices

As mentioned above OSGi provides functionality for replacement of OSGi bundles while a system remains running. The advantages of this functionality are especially apparent for scenarios in which long running processes exist. For instance, a long running process could be controlling an industrial machine. For such a situation modular updates or changes to the software can be valuable.

However, the expected deployment within the platform designed in this work is slightly different. Instead of long running processes the system is expected to be deployed to cloud environments. Here, virtual machines can be bootstrapped or shut down quickly. The deployed applications play a small role in the entire system. In this situation long running systems are

[2]`https://cxf.apache.org/distributed-osgi.html`
[3]`http://www.aqute.biz/Bnd/Bnd`
[4]`http://bndtools.org/`
[5]`https://developer.ibm.com/wasdev/downloads/`

less likely to occur. An alternative to updating running systems is the gradual re-deployment of new versions. In this situation multiple servers running some version of an application exist. By gradually swapping these instances with upgraded versions the system can be updated while the platform remains accessible.

In combination with upcoming containerization technologies this approach has several advantages over the OSGi approach. Through these containerization technologies immutable system images can be created, thereby ensuring that two machines running the same image run with exactly the same configurations. This is in contrast to the OSGi approach in which processes run for longer durations. In this approach it is much harder to ensure that multiple instances running the same software have exactly the same configurations. This becomes even more complicated when systems are gradually updated over time. Changes between configurations of machines running the same software is known as configuration drift.

OSGi conclusions

Concluding, there are numerous advantages and disadvantages to using OSGi for the platform designed in this work. The points described in the previous sections explain some of the motivations behind not using OSGi as a development platform. The main reasoning can be summarized as that the various development tools for OSGi in combination with the WebSphere Liberty tools did not integrate fluently and that the capabilities of OSGi were not likely to be utilized fully in the resulting implementation. Therefore, other technologies have been used for the proof of concept implementation. However, the lessons learned from the modular approach of OSGi have certainly helped in designing a modular system. For instance, in place of the dependency management capabilities of OSGi a dependency injection framework was used.

6.2.2 Hardcoding core component network locations

In the REST based distributed architecture of the system the network locations of core components are essential. Most of these locations can be stored in the Application repository. For the Applications repository however the network location was hardcoded in the Central API. During the discussion of the implementation this point has come up. An initial solution to this problem is the use of environment variables for seeding Central APIs. Additionally, Docker container networking[6] has been utilized for communications between core components. Instead of hardcoding networking locations in the form of IP addresses Docker networking allows components to be reached on specific domains. For instance, the Applications repository might be reachable on the address *https://applications*. The application repository is then bootstrapped with these initial network addresses. Although this is a step up to the initial solution, this approach is not yet prepared for more advanced scenarios in which replication and fault-tolerance are desired. With this approach a more extensive service discovery infrastructure might be needed.

[6]`https://docs.docker.com/engine/userguide/networking/dockernetworks/`

6.2.3 Scalability of the Authentication component

As previously mentioned a challenge with the current implementation is the scalability of the Authentication component, which was questioned during the technical discussion. Since the Authentication component uses asymmetric cryptography for signing JWTs, replication of this component means that private keys have to be shared. This means that the component is not scalable in its current form. To combat this issue a more elaborate implementation is required. Such an implementation can make use of specialized cryptography hardware to ensure the security of private keys. Alternatively every JWT can include an identifier that specifies which instance has signed it. This way, every Authentication component can keep a private key in memory, subsequently JWTs are verified using the public key of a specific instance. A downside with this approach is that when a single Authentication instance fails the JWTs issued by this instance can not be verified anymore, requiring another proof of identity for these sessions. Concluding, additional efforts have to be made to ensure scalability of the Authentication component.

6.2.4 Node administration through HTTP headers

During the technical discussion the point was raised that node administration does not necessarily have to be explicitly performed by third party servers. Instead, HTTP headers can be inspected to decide whether any changes have occurred to the graph model. Through this optimization the usability of the platform has increased greatly. Application developers can use existing practices in terms of HTTP response headers to specify any changes to the graph model. These practices are likely already implemented in most REST interfaces and thus this would introduce minimal effort on the application developers part for making full use of the platform's functionalities.

6.2.5 Implementation discussion conclusions

The technical discussion raised some interesting questions with regards to the design choices for the implementation. Some of these points remain valid and in future work some of the decisions could be revisited to determine whether a better solution can be constructed. Nevertheless, the current implementation provides a guideline for a future improved version. This implementation can either be expanded upon or used as an inspiration.

6.3 Scalability

As already discussed scalability is an important requirement of the system and therefore scalability has been kept in mind during architecture design. In order to verify the fulfillment of this requirement several tests have been conducted. Performance of the system has been analyzed in different ways. First of all, since the system is built out of multiple stand-alone core components, performance is analyzed on a component level. For each component, there are multiple interesting characteristics. Besides individual performance the next point of analysis is the system as a whole. Besides testing core components another important characteristic of the system is how it proxies requests to third party servers. In order to test the effects of different characteristics of these third party servers on overall performance a test server has been constructed. This test server responds differently to requests based on query parameters. Varying factors include the number number of permissions and the payload size.

Before discussing the results the expected platform load is discussed. This is followed by some notes about different scalability characteristics and about the testing approach.

6.3.1 Expected load

The platform designed in this work is targeted towards enterprise environments. In these environments several offices exist that work together remotely. The scope of the platform is an important factor in determining the expected traffic load. As an example several scopes can be identified; Assume that a single office houses roughly 3000 employees. Then assume around 6 offices exist in a single country. Thus, in a single country there are roughly 18.000 employees. Besides this, an international scale could be much larger, such as the 377.757 employees working for IBM worldwide[7].

When dealing with employee data these different scopes play an important role. Some enterprise applications deal with data mostly confined to a single office. For instance, an application that allows employees to schedule office space in a certain building only requires data for that specific office. For other applications there are more opportunities when data gathered from multiple different regions is combined, such as for instance statistics about office climate. For such a scenario the main advantages involve large scale analytics. For other applications such as the gamification example application it can be engaging to see scores of close colleagues, but seeing the results of other offices might also be beneficial for building a global company culture.

Keeping these scopes in mind, the throughput requirements of the platform could become very high. Since the platform acts as a central gatekeeper for possibly a large amount of applications there could be a substantial throughput that needs to be handled. In return however application servers require less administrative functionalities. When looking at a single office, consider a scenario in which every employee uses several applications on their smart phone. Assume that combined these applications make a request every 10 seconds. This would result in 300 requests per second or 8.640.000 requests during an eight hours work day. This of course depends a lot on how applications utilize cloud services. Nevertheless, this is already a substantial load for a single office. When using the same assumptions a single country would generate 1800 requests per second or 51.840.000 requests during an eight hour work day. Finally, when looking at IBM as a whole, these assumptions would lead to roughly 12.592 requests per second worldwide. Note that these numbers are very simplistically calculated and do not have any substantial or definite meaning. Actual numbers would depend heavily on the usage patterns of enterprise applications. Such patterns could be studied in more detail in future work.

6.3.2 Testing environment

Since each core component is a stand-alone application they can easily be deployed to cloud environments such as IBM SoftLayer[8]. By using such an environment the stateless nature of

[7] `http://www.ibm.com/investor/att/pdf/IBM_Annual_Report_2015.pdf`

[8] `http://www.softlayer.com/`

these core components can be utilized to scale the system. Since this strategy is a key part of the system design the scalability testing has been done in a cloud environment. To ease the process of testing as well as deployment several tools were used, which will be discussed in the next sections.

Testing scenarios

During testing two scenarios were used, each having a specific purpose. First off all several tests were performed on low end hardware, which will be referred to as the *low end scenario*. The main purpose of these tests was to find the relative performance of each core component, as well as the impact of various implementation specific factors. These results are useful as the slowest core component determines the performance of the overall system. Secondly several tests were performed using higher end hardware, which will be referred to as the *high end scenario*. In these tests the system as a whole was tested. The purpose of these tests was finding the capabilities of the system under heavy load. The results of these tests can be used to determine the applicability of the solution to an enterprise environment.

Virtual machines

As a basis for deployment virtual machines in a cloud environment are used. For the low end tests machines with 512MB RAM and 1 CPU are used for deployment of core components. Additionally three machines with 2GB RAM and 2 CPUs are used as a Cassandra cluster. For the high end tests machines with 16GB RAM and 4 CPUs are used. All machines run Debian 8 x64[9].

For ease of deployment each machine runs Docker Engine[10]. Docker Engine is a containerization technology that can be used for deploying light weight containers to machines. Provisioning is done using Docker Machine[11]. Docker Machine is a tool for installing and managing Docker Engine on virtual hosts.

Clustering

In order to experiment with scalability characteristics of the system Docker Swarm[12] is used. This technology provides clustering capabilities for several machines running Docker Engine. By setting up a Docker Swarm various machines running Docker Engine can be exposed as a single virtual Docker Machine. For setting up a Docker Swarm three types of machines are configured. First off a single machine runs Consul [13]. Consul is a distributed service discovery service. Secondly a Docker Swarm master is set up. Finally additional machines running Docker Engine can be added to the swarm. Provisioning is done for these different types of machines through Docker Machine.

[9]https://www.debian.org/
[10]https://docs.docker.com/engine/
[11]https://docs.docker.com/machine/overview/
[12]https://docs.docker.com/swarm/overview/
[13]https://www.consul.io/

Docker compose

Docker Compose[14] is used to deploy the implementation. This tool can be used to deploy multiple docker containers to a Docker Machine at once. In combination with Docker Swarm this tool can thus be used to deploy containers to multiple virtual machines. Additionally, Docker Compose provides convenient utilities for setting up networking between Docker containers.

Docker containers

For deployment of the implementation several docker images are used. First of all the official Cassandra Docker image[15] is used (v3.3), which acts as the main storage technology. Secondly, an Nginx[16] image[17] is used for setting up HTTP load balancers for some of the testing set ups. Nginx is, among other things, a reverse proxy. For each of the core components a docker image is created using the official WebSphere Liberty docker image[18] as a basis. Every image contains a WAR file with the required Java class files and third party dependencies.

6.3.3 Network topology

An important part of deployment is the network topology. This topology determines which applications run on which virtual machines. During testing several topologies are used, an outline of which will be discussed in the next sections.

Single components

For testing single components two approaches are used. In the first approach a single virtual machine is used for running a component, as shown in Figure 6.1. This topology can be extended by replicating several instances of the component and putting these replications behind an Nginx instance serving as a load balancer. This approach is shown in Figure 6.2. As previously mentioned replication is possible due to the stateless nature of the core components. For this setup a round-robin distribution strategy is used, meaning that requests are proxied in turn to each component instance. Additionally core components that require persistence use a Cassandra cluster of three nodes.

[14]`https://docs.docker.com/compose/overview/`
[15]`https://hub.docker.com/_/cassandra/`
[16]`http://nginx.org/`
[17]`https://hub.docker.com/_/nginx/`
[18]`https://hub.docker.com/_/websphere-liberty/`

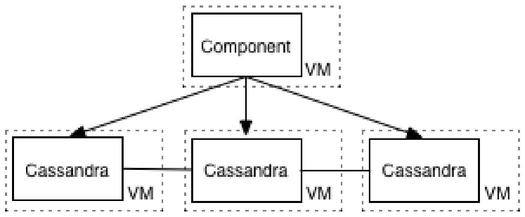


Figure 6.1: Network topology single component (low end)

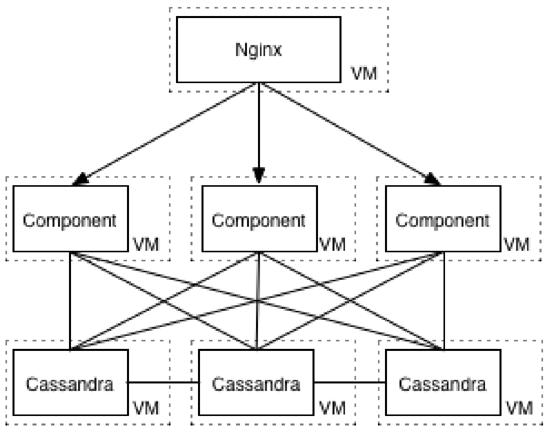


Figure 6.2: Network topology replicated components (low end)

Combined components

For the combined components there are a lot of possibilities with regards to network topology. For the low end testing scenario each component is deployed to a single virtual machine as shown in Figure 6.3. Note that the authentication component and applications repository component are deployed to a single virtual machine. This is due to these components playing a small role in the proxying process considering that the data needed from these components is cached by the central API. Additionally a second approach for the low end testing scenario is shown in Figure 6.4. In this approach two replicated Central API components are used, load balanced by an Nginx instance. Note that in this approach the Cassandra cluster is reduced to two instances due to limited testing capacity.

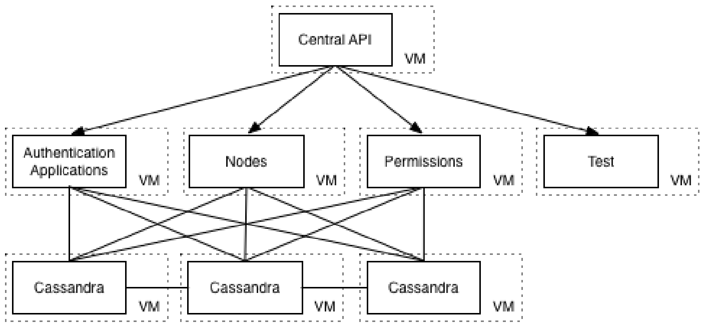


Figure 6.3: Network topology combined (low end)

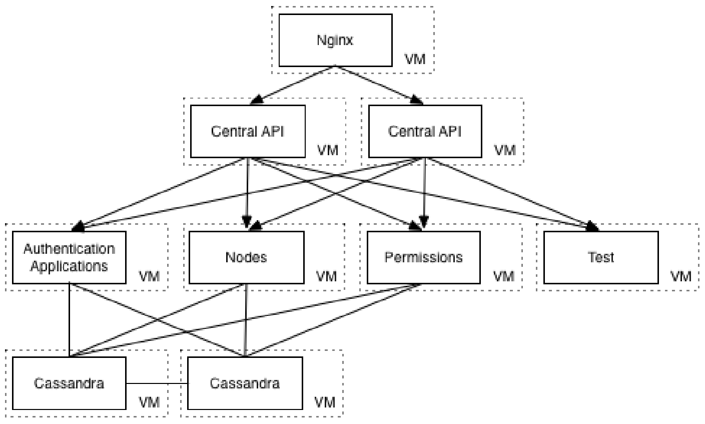


Figure 6.4: Network topology combined with replicated Central API (low end)

For the high end scenario the approach was slightly different. In this scenario each VM had substantial more resources. Initial tests showed that with these extra resources CPU utilization was rather low. To utilize these resources more effectively multiple components were deployed to single virtual machines. The network topology for this approach is shown in Figure 6.5. Note that this setup is purely experimental and more efficient deployment approaches might be possible. For instance, it might be better to run Cassandra on dedicated nodes. For more comments on performance see subsection 6.3.10.

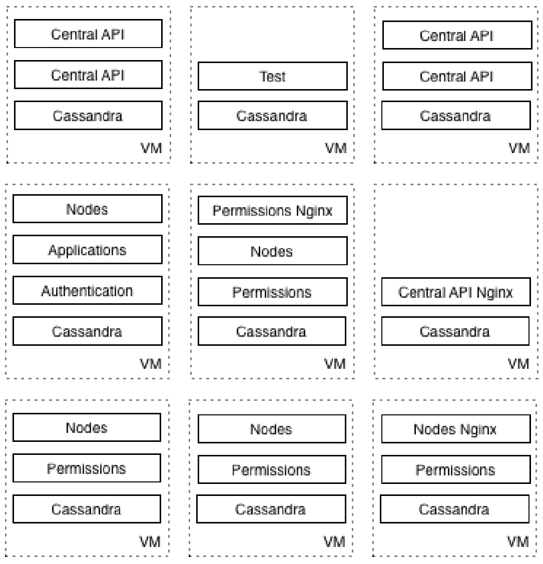


Figure 6.5: Network topology combined (high end)

6.3.4 Testing tools

For the low end scenario testing was done using Apache JMeter[19]. Several test plans were executed, each testing different properties of the system. For these tests a *Throughput Shaping Timer* plug-in was used. This plug-in provides functionality for specifying the desired throughput at which requests are made. Using this approach the limits of the components can easily be determined. When these limits are reached, response latencies increase as the system becomes saturated. During tests discussed in this section this approach is used to determine the maximum throughput of each core component. For each test case the number of transactions is determined to a point where latencies are influenced. This number of transactions is calculated numerous times and averaged. JMeter tests were performed in GUI mode on a MacBook Pro with a 2.9 GHz processor and 8GB RAM.

Although the tests performed with Apache JMeter provided some valuable insights into relative performance characteristics the results are not completely realistic. First off all running JMeter in GUI mode provides additional processing overhead, thereby reducing the available resources for actual requests. Additionally, considering that every requests originates from a single machine, a similar network path is used for each request. This results in a high impact of network performance in tests results. Therefore, for testing the high end scenario Loader.io [20] was used. Loader.io is a cloud-based load testing service that uses the Amazon Web Services (AWS) platform to generate load. By using this service the results are less biased towards local

[19] `http://jmeter.apache.org/`

[20] `https://loader.io/`

network performance. The results of the Loader.io tests provide insights into expected latency of the system as a whole.

6.3.5 Testing the Authentication component

The authentication component's main responsibility is token management. The component has responsibility over every aspect of authentication tokens, namely constructing JWT access tokens, storing refresh tokens and exchanging refresh tokens for access tokens. Due to the caching implemented in the central API overall load of this component will not be substantial compared to other components. Therefore this component will mostly be used to construct new JWTs.

Creating JWTs

The JWT creation service translates an authentication context into a signed JWT. This service is called when users log in and, considering the short validity duration of the JWT tokens, additional times during application usage. Therefore, the expected load for this system is significant, although less than for the data repositories. Testing this component with increasing throughput shows an average bound of *280* transactions per second, after this latencies increase. The results of a single test are shown in Figure 6.6. As can be seen from this figure the number of transactions per second becomes irregular at the end of the test which is paired with an increased response latency.

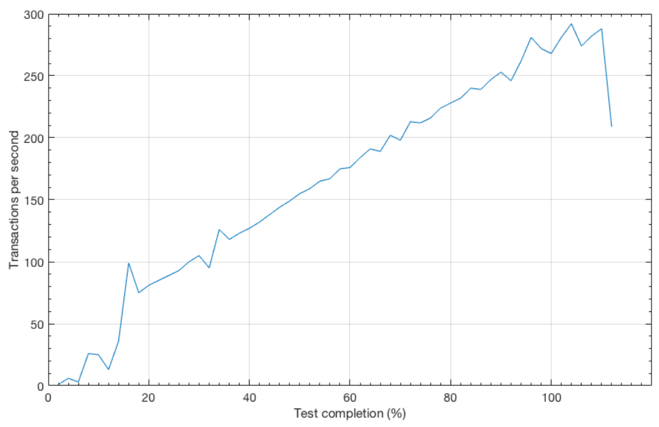


Figure 6.6: Creating JWT tokens, increasing transactions per second

Verifying JWTs

Besides creating JWTs the Authentication component also provides a service for verifying JWTs. This service takes a JWT as a parameter and determines its validity by verifying the signature against the key that is used to sign tokens. Note that this functionality is not used in the proxying process due to an optimization. In this optimization the Central API caches the public key exposed by the Authentication component and uses this public key to verify JWTs directly. Nevertheless results of testing this verification service provides an upper bound on the number of requests that can be processed by the Central API, since this services performs a subset of the computations done during the proxying process. Testing this service resulted in an average of *750* transactions per second before latencies are influenced.

6.3.6 Testing the Applications component

The application repository component as previously stated is in charge of storing the application model. However, it also plays an important role in the proxying process. The applications component is responsible for looking up application meta data needed during the proxying process. This meta data is also cached in the central API. Nevertheless, lookups need to be fast, even when a lot of requests are being handled. Since the number of applications in an enterprise is relatively low the amount of available meta is not expected to grow as fast as data in the other repositories.

The performance of this component scaled from one to three instances is shown in Figure 6.7. As this figure shows a single instance of the Applications repository handles slightly less than *300* requests per second. When adding additional instances the number of requests increases by roughly *125* for each instance.

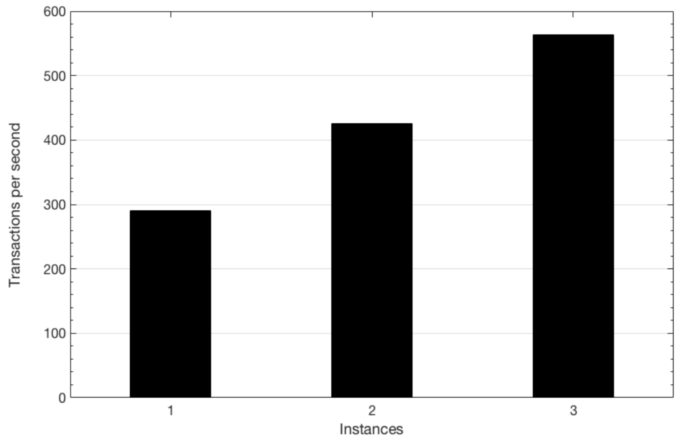


Figure 6.7: Retrieving applications, max transactions per second

6.3.7 Testing the Nodes component

The node repository plays multiple important roles in the proxying process. First of all, the repository is extensively used for node management. When node states change during service invocations, the nodes repository is used for storing these changes. For instance, when nodes are created during a service invocation, the nodes repository must store their existence, as well as relational information about ownership. Besides storing node meta data, the nodes repository is also extensively queried during rule evaluation. Since rule validation equates to relationship existence verification, the nodes repository must be able to quickly answer such queries, even when the number of rules increases. Concluding, both the number of concurrent requests and the amount of rules queried at once are important varying factors.

Creating nodes

The node creation services creates nodes and initializes relationships. This service is used by the Central API when new nodes are created through external services. During node creation a single node entry is made as well as three initial relations; one describing the identity of a node (*nodes:is*), one describing the ownership relation (*nodes:ownedBy*) and finally one describing the reverse of the ownership (*nodes:ownerOf*). These initial relationships are used in default permission rules.

The resulting average maximum transactions per second for varying instances are outlined in Figure 6.8. Additionally Figure 6.9 shows the number of created nodes and relations per second. As these figures show, the number of requests per second starts at roughly *240*. For each addition instance the throughput increases by roughly *150*.

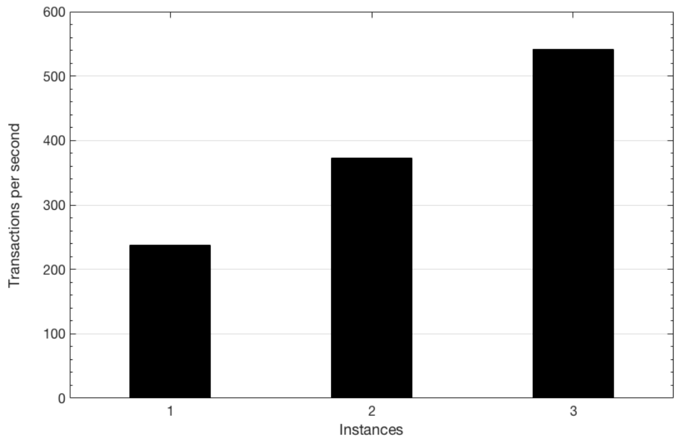


Figure 6.8: Creating nodes, max transactions per second

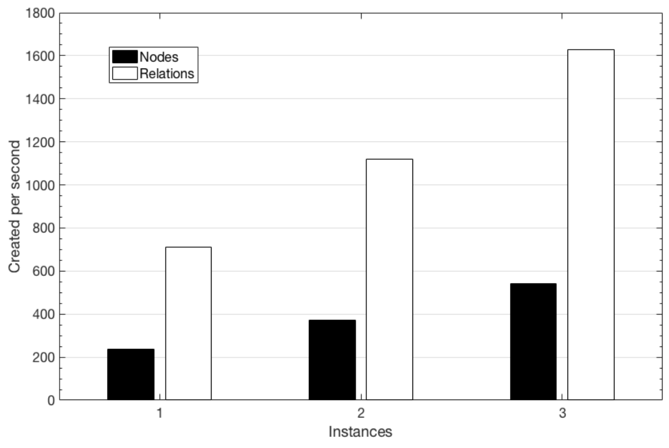


Figure 6.9: Creating nodes, max created per second

Querying relations

During proxying the central API determines the relevant permissions rules that apply to a request. These permission rules state relationships that must exist in order for the access to a service to be accepted. For determining existence of relationships a single service is available that accepts a list of possible relations and determines whether any exists. For these queries an important factor in the performance is the number of rules per permission, considering the $O(n)$ complexity of rule evaluation. In order to analyze the effects of an increasing number of rules the service is invoked with varying number of relations.

The average maximum number of transactions with the varying factors are outlined in Figure 6.10. As can be seen in this figure the number of queried relations has a substantial impact on performance. Especially when 100 relations are queried at once the number of transactions per second is very low. When the component is scaled to additional instances the capacity increases roughly linearly. A single instance starts with an average maximum of *140* requests per second. This increases to *280* with two instances and *355* with three instances. In Figure 6.11 the total number of queried relations are shown. As this figure shows the even though the overall number of transactions per second decreases, the total number of queried relations increases linearly.

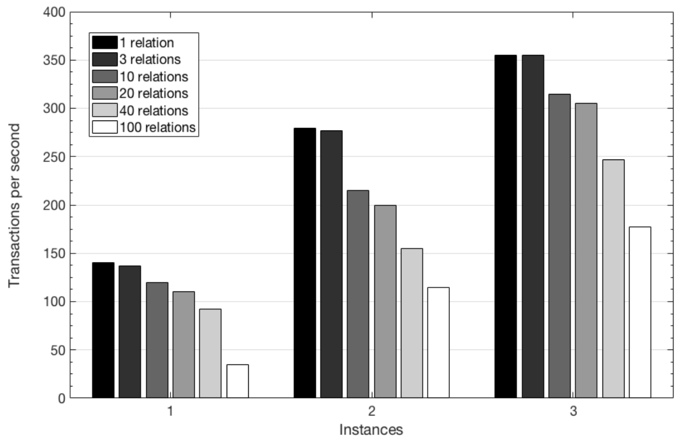


Figure 6.10: Querying relations, max transactions per second

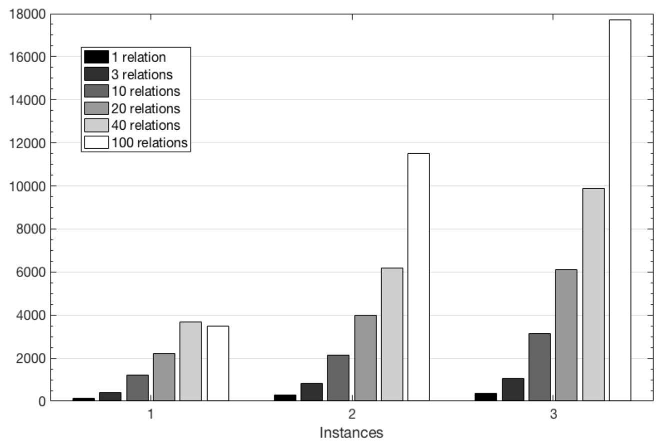


Figure 6.11: Querying relations, max relations queried per second

6.3.8 Testing the permissions component

The permission repository also plays multiple roles in the proxying process. During node creation, default permissions for nodes have to be created in the permission repository. The challenge with this responsibility is that the number of default permissions could be high. Besides creation of permissions the repository is also responsible for quickly looking up permissions for certain services and service consumptions. These lookups are a key part of the proxying process.

Instantiating default permissions

The permission repository provides a service for instantiating the default permissions paired with a specific service and service consumption. This service accepts the details of a node and the targeted service or service consumption and uses this information to determine the default permissions that have to be created. A varying factor in this is the number of default permissions paired with the targeted service or service consumption.

Figure 6.12 shows the results for this test. As can be seen from this figure the number of default permissions has a significant impact on the maximum number of transactions, although less than what was seen in the relations query service. A single instance starts with an average maximum of *160* requests per second. This increases two *325* with two instances and *400* for three instances. The average maximum of the total number of permissions created are outlined in Figure 6.13. This figure shows the same pattern as for the relations query service, namely that the data throughput increases linearly while the total number of transactions decreases.

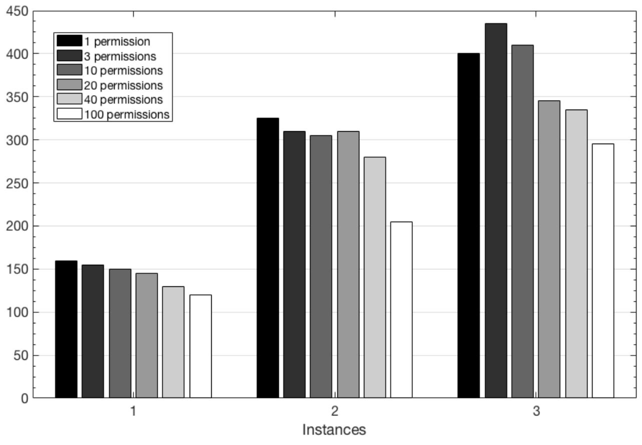


Figure 6.12: Instantiating default permissions, max transactions per second

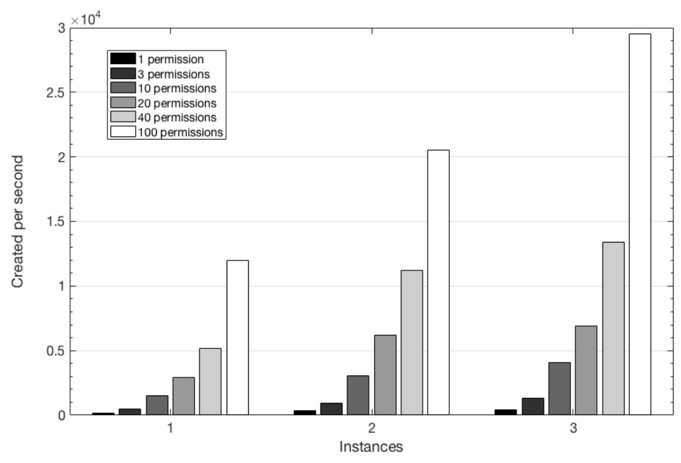


Figure 6.13: Instantiating default permissions, max created permissions per second

Querying permissions

During request proxying a key part of the access control decision are the relevant permissions. For each request two permissions can apply, one for the service and one for the service consumption. This means that for each requests two permissions have to be retrieved. Figure 6.14 shows the average maximum number of transactions per second for respectively one, two and three instances. As this figure shows a single instance has an average maximum throughput of *250*. This increases to *490* for two instances and *660* for three instances.

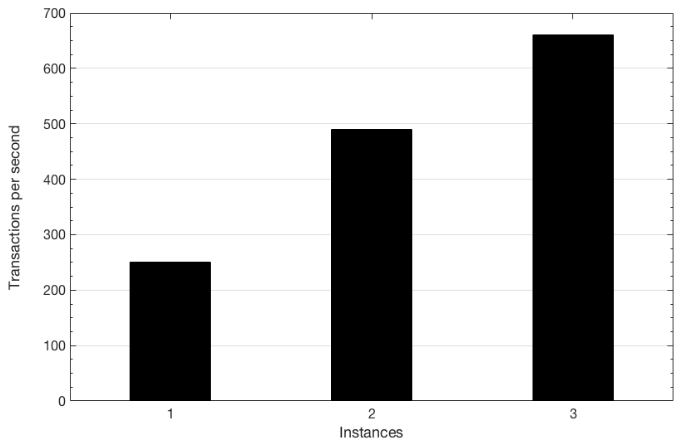


Figure 6.14: Querying permissions, max transactions per second

6.3.9 Testing the Central API

As previously mentioned the Central API proxies requests to third party servers. During this proxying process several core components are invoked to determine whether access is allowed. Furthermore additional administration is done to keep the Nodes repository up to date. Due to the dependency on core components the performance of the Central API is bound by the performance of the core components. Specifically the slowest core component provides an upper bound on the throughput of the central API. By using the results from the tests targeting individual components an upper bound on throughput of the Central API can be determined.

For testing the central API a single standalone test server was used. This test server exposes two services through the Central API, one retrieval service and one creation service. For the retrieval service a query parameter can be provided that determines the amount of data returned in Bytes. The creation service accepts an arbitrary payload size. Additionally, several permissions and default permissions were created such that these factors could be varied during testing. As was mentioned earlier two tests scenarios were performed, a low end scenario and a high end scenario. The next sections detail the outcomes of tests under these scenarios.

Testing the Central API: Low end scenario

In the low end testing scenario the goal was to determine the performance of the Central API relatively to the other core components, namely the Nodes repository and the Permissions repository. Additionally the payload size is varied to determine performance of the proxying itself.

Figure 6.15 shows the performance with network topology 1 (Figure 6.4) and network topology 2 (Figure 6.3). As this figure shows the number of request per second is relatively low. With network topology 2 the performance increases, but not substantially. The reason for this low number of request might be coupled with the fact that in these network topologies there is only a single Nodes repository instance. As shown in Figure 6.3.7 a single instance of the nodes repository has a capacity of *135* requests per second on average. Since for each request this service is invoked the throughput of the Central API is limited by this number. The figure also shows the impact of an increased payload size of API requests, which shows to have some effects on overall throughput.

Figure 6.16 shows the performance with the same network topologies but instead for the creation service. A key difference between these service invocations is that for the creation service additional administrations are performed to keep the Nodes repository up to date. As these figures show this additional administration has substantial impact on overall performance of the Central API.

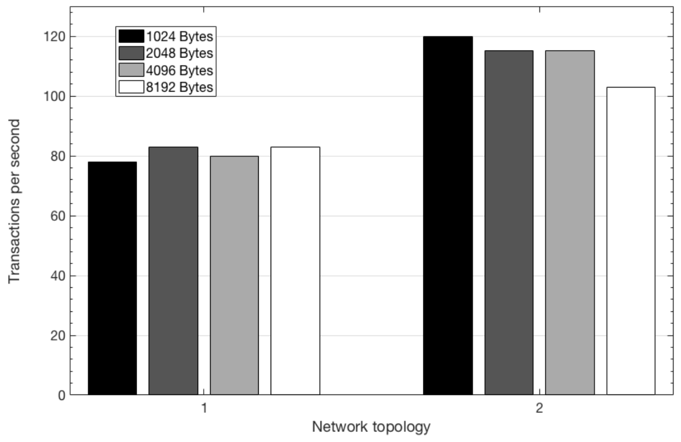


Figure 6.15: Central API retrieval test, max transactions per second

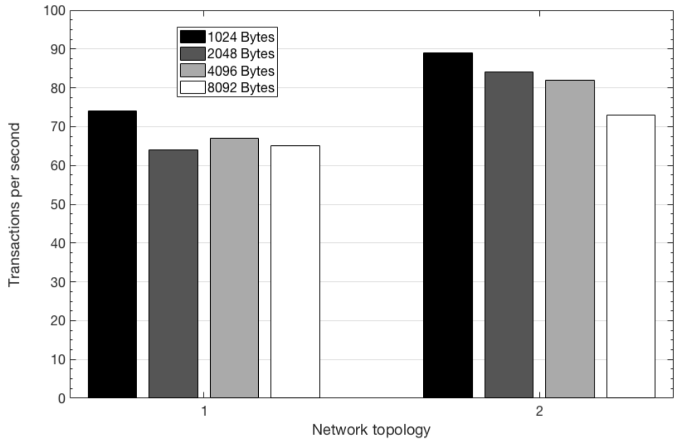


Figure 6.16: Central API create test, max transactions per second

Testing the Central API: High end scenario

Although the tests under the low end scenario provide some insights into performance of the Central API under different circumstances, the resulting throughput is rather low. To determine whether the performance can be improved by using better hardware the high end scenario was used. Additionally Loader.io was used for performing test to reduce the effects of local network performance and other influences on test results. Loader.io offers the functionality to configure the desired number of clients per second during a specific time period. Several different number of desired clients per seconds were used to determine the effects of higher load on Central API latency as well as the maximum throughput.

For these tests two versions of the network topology as given in Figure 6.5 were used. In one version two instances of the Nodes repository and the Permissions Repository were used. After these tests showed relatively low CPU utilization under heavy load additional instances of each repository were deployed to the same number of machines. Thus, for each repository four instances were used behind a single Nginx load balancer. The effects of these additional instances are substantial. For the first version of this topology the maximum number of requests per second without effecting latency is on average *525*. During these tests the average latency was *130 ms*. The same test applied to the second version of the topology results in an average maximum of *600* requests per second at a latency of *125 ms*.

Figure 6.17 shows the average maximum number of transactions above which latencies are influenced significantly. This figure also shows the effects of the number of permission rules available for the requested node. As can be seen the number of transactions is substantially higher than during the low end scenario. Nevertheless, the number of applicable permission rules has a substantial impact on performance. The average response latencies are shown in Figure 6.18. As this figure shows the response latency also increases as more applicable permission rules are available.

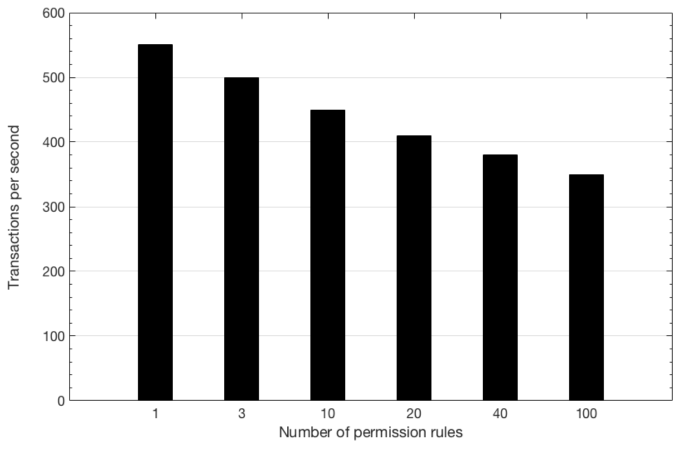


Figure 6.17: Central API retrieval test, max transactions per second (high end)

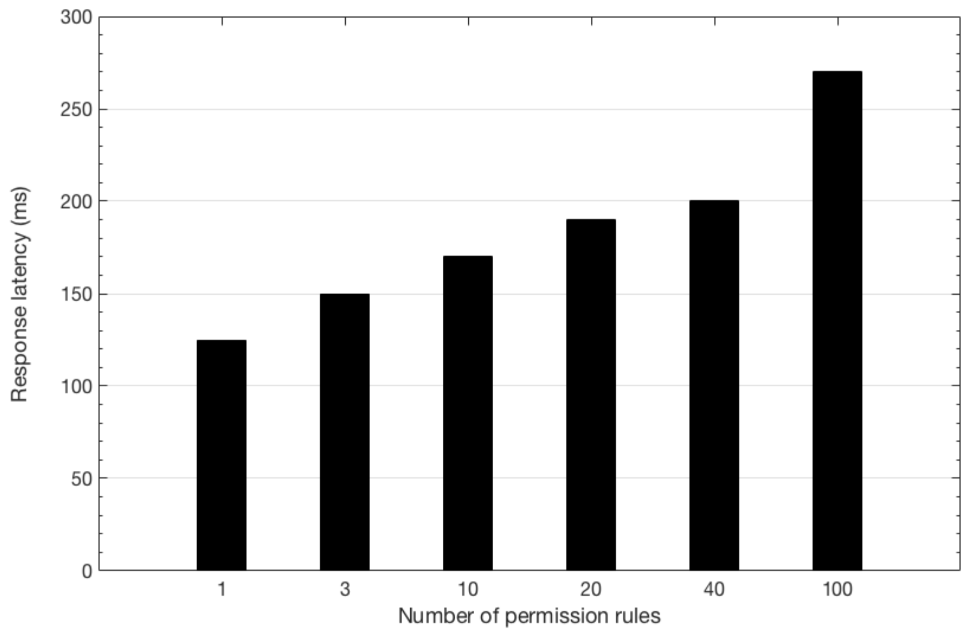


Figure 6.18: Central API retrieval test, average response latency (high end)

6.3.10 Discussion of performance

The test results discussed in the previous sections have some implications regarding the usability of the implementation. In this section the results will be discussed.

Reflection on expected load

The numbers given in subsection 6.3.1 provide some initial targets in terms of required throughput of the platform. During the tests in the high end scenario realistic throughputs of 400 to 550 requests per second were achieved. With regards to the expected load this would mean that only a single office can be sustained during heavy application usage by employees. This indicates that there is still work required in terms of implementation optimizations before wide scale adoption is possible. Additionally, the latency during realistic test scenarios under heavy load was roughly $125ms$ to $175ms$. Depending on third party application requirements this could be too high, especially when third party services are computationally heavy by themselves. For these scenarios additional optimizations might be needed. In the next section possible optimizations are discussed.

Performance gains

Although the implementation falls slightly short in terms of expected load, there are some promising test outcomes in terms of scalability. Since the platform is built on top of scalable technologies there are opportunities for horizontal scaling. This scaling behavior can be seen during the tests targeting individual components as well as the overall platform. However, before such scaling is used it might better to optimize the implementation first. There are several points at which the implementation could be optimized. These optimizations have not been implemented due to time constraints. The next paragraphs discuss several potential optimizations.

JVM Profiling and tuning A common observation during testing shows that all core components experience occasional hiccups. During these hiccups response latencies peak, resulting in an increase in latency of roughly $50ms$. The cause of these hiccups could be related to Java garbage collection. In order to gain more insights into the garbage collection behavior it might be worthwhile to profile the core components. During profiling insights can be collected to determine the performance of specific code segments. During this profiling potential issues can be detected. These issues might include code segments that produce a lot of garbage, leading to more frequently triggered garbage collections. Furthermore, currently the JVM has not been tuned at all. This might be another point of optimization.

Limitations on the data model As the test results given in Figure 6.3.7 and subsection 6.3.8 show, the number of permission rules and default permissions have a significant impact on performance. This behavior is also observable in the high end scenario (Figure 6.17), in which the maximum throughput is heavily influenced by the number of applicable permission rules. Although this might be an issue it should be noted that *100* permission rules or default permission are significantly more than what is expected during normal usage.

Consider the image uploading service to which users can upload images and consequently determine who can view these images. Consider also an application that manages social relations between users. This social application might define relations such as *social:friendOf* or *social:familyOf*. It is expected that the permission rules for the service exposing images only target a single relation such as the onces named above. Thus, instead of requiring a single rule for every user who requires access to the image a single permission rule can be used to describe a larger group. When individual access is required it could also be an option to create an application that manages user lists. This application could allow users to create custom lists and subsequently add users to this list individually. The fact that a user is part of a list can be documented in a *lists:partOf* relation. This relation can then be used in a permission rule. This is analogous to the privacy settings on existing social networks, where the options include settings such as *public* or *friends only*. By using this approach the required number of permission rules is reduced drastically. It could therefore be realistic to impose an upper bound on the number of permission rules allowed for a single permission. Based on the test results such an upper bound might be set to a number between 5 and 10. This would still provide a lot of configuration opportunities for users while also providing relatively high performance.

A similar expectation holds for default permissions. When nodes are created the default permissions are expected to be focused towards access through a single application and mainly by the owner of the created node. For example, when an image is created in the image uploading service, the default permissions might provide the owner with the ability to manage the image. Thus, only three default permissions could be needed, namely for retrieving, updating, and deleting the image. Additional permission rules can be added by the owner of the image afterwards. Considering this expected usage pattern a similar upper bound can be applied to the allowed number of default permissions.

Hardware configuration Although the deployment used a relatively large number of virtual machines, several signs indicate that the hardware is not fully utilized. First off all, network traffic was not even close to saturation. Secondly, during the tests in the high end scenario each VM had a maximum load of 40%, while most VMs leaned towards a much lower CPU load of roughly 20%, even when multiple Docker images were running on the same host. Finally, disk activity was also rather low.

A reason for such inefficient resource usage might be due to the fact that a lot of configurations use default values. These configurations target things such as the JVM, Cassandra connection pooling, internal HTTP communication connections, Java thread pooling, Nginx load balancers, VM configurations, kernel options or Docker configurations. In order to determine the current bottlenecks in the implementation more experiments are needed. During future experiments it would be desirable to determine whether the implementation in its current form is network bounds, CPU bound or disk bound. Further optimizations can then be made accordingly.

Finally, in the testing environment no distinction was made between virtual machines, even though different parts of the implementation require different resources. For a more solid deployment it might be better to deploy subcomponents to hardware that fits their use case best. For instance, the Cassandra cluster could be deployed to dedicated hardware with carefully chosen disks. Future experiments can aim to determine the resource requirements of individual subcomponents more accurately.

Internal communication protocol Currently communications between the Central API and core components is done over HTTP. This approach provided a number of advantages; first of all the applicability of the model could be tested directly for these core components. This provided several insights into the capabilities of the model. Additionally, frameworks such as JAX-RS could be used for quick development. Although these advantages have certainly helped during the implementation, there are some downsides to this approach, the main downside being performance. Without any additional configuration the HTTP protocol establishes a new TCP connection for each request. This is obviously not desirable for internal communications since it introduces an unnecessary overhead. A possible optimization could be to use HTTP persistent connections. This would have to be configured in both WebSphere Liberty and the Nginx load balancers. An alternative could be to implement support for additional communication protocols that better fit the job.

Lower level networking for proxy requests Currently the implementation of the Central API also makes use of JAX-RS for proxying. Although this provides a basic implementation it might be possible to achieve better performance by making use of lower level networking libraries.

Caching Currently no caching is used besides for the application meta data in the Central API. Additional caching could be implemented for the Nodes repository and the Permissions repository. These components could make use of a cache such as Redis[21]. Whether caches are applicable on a large scale for these components remains an open question however. Since these repositories are expected to contain a lot of data, cache hit ratios could be very low. On the other hand, the data stored in these repositories is very small, thus it could be possible to load a large part of the data in memory. Such a caching layer does introduce additional complications in terms of cache invalidation.

6.4 Fulfillment of the requirements

In chapter 2 several requirements were discussed that are essential for a privacy management solution within an enterprise. In this section these requirements are discussed in context of the resulting framework.

6.4.1 Non-functional requirements

The non-functional requirements as given in section 2.4 provided a set of high level goals that required additional attention during the design phase. Even though these requirements are not very concrete some comments can be made on their fulfillment. An important note here is that while these non-functional requirements have been kept in mind during the design phase, it is likely that there will always be improvements that increase the fulfillment of these requirements even further.

***NFR.1* Secure** The designed architecture is a central point in terms of security. Because the platform acts as a central proxy between various actors with the goal of protecting private data there is a lot of responsibility in terms of security. The model itself provides users with the security that access to personal data is limited to those that have obtained the correct permissions. However, a lot of this security is dependent on the correctness of the implementation. To ensure solidity of the implementation a simple model was designed and existing and proven standards as well as high quality tools were used. Nevertheless security remains an important point of attention and additional security measures might have to be introduced to ensure overall system security.

***NFR.2* Modular** The application model provides a framework for modular applications that work together towards the goal of a more deeply integrated working environment. The framework aims to impose little restrictions on implementation details while carefully orchestrating dependencies between different applications. Even though the basis for a modular environment

[21]`http://redis.io/`

is constructed, the applicability of the data model does impose some restrictions on the implementations. Namely, the ownership based model introduces several restrictions on API design. This could impose challenges when connecting existing services to the platform. Even though this model provides a simple way to handle privacy preferences across systems it remains a point of future research whether this model is too restrictive.

NFR.3 **Scalable** As initial experiments show the scalable architecture of the system seems to provide opportunities for large scale deployment. Even though these initial results fall short of a direct nation wide deployment under heavy load, there are several possible improvement points. Nevertheless scalability should be kept in mind in future work because of its importance in enterprise environments.

NFR.4 **User centric** Throughout this work the non-functional requirement of user centrality played an important role. The model is heavily focused to individual control over privacy settings and thus users are a centric part of the resulting framework.

NFR.5 **Implemented using tools compliant with IBM guidelines** By implementing a solution on top of IBM technologies a lot of the compliance requirements were fulfilled directly. In addition several open source projects were used for the proof of concept implementation. All of these solutions were either approved by IBM or easily replaceable by approved alternatives. Using this approach compliance with the IBM guidelines was ensured.

6.4.2 Functional requirements

The fulfillment of the non-functional requirements is partly subjective. The functional requirements given in section 2.5 however provide some guidelines as to how the results implement the predefined functionalities. The next sections discuss these requirements.

AU.1 **Identifying users in different contexts (✓)** The use of the JWT standard provides functionality to handle users in different contexts.

AU.2 **Revoking previously successful authentication attempts (✗)** By using short-lived signed access tokens in combination with long-lived persistent refresh tokens revokal of successful authentication attempts can be implemented. Although the design and used standards allow for this functionality, due to time constraints only short-lived access tokens have been implemented.

PP.1 **Extensible specifications of data handling practices (✓)** The ability to specify data handling practices through a standardized specification language is included in the application model through service consumption policies. Using this model every service consumption can specify a single policy that describes the relevant practices regarding data handling. The specification language however remains undecided in this work. The reasoning behind this is that the goals of this work do not include the invention of a new policy specification language. Rather, existing policy specification languages can be utilized. In this regard the results of this work provide a framework for more closely integrating existing policy specification languages in a multi-tenant application environment.

***PP.2* Specification of data usage restrictions (✓)** Specifications of data usage restrictions can be specified through service policies. For these policies the same reasoning applies as given above; rather than inventing a new policy specification language existing languages can be utilized in the framework designed in this work.

***PP.3* Describing different purposes of data usage (✓)** The description of different purposes of data usage is included in the application model through service consumption identifiers. Through this construct applications can specify multiple reasons for accessing data.

***PP.4* Infrastructure for dealing with policy changes (✓)** A core part of the application model is versioning. Service and service consumption versions are inherently linked to policy updates and thus an initial infrastructure is made that allows applications to deal with such changes. The versioning system also supports service deprecation, meaning that services can be flagged as being deprecated. Through this infrastructure changes to an application can be made apparent to other developers depending on the involved services.

***PP.5* Infrastructure for dealing with policy changes by external data sources (✓)** The infrastructure used to deal with internal policy updates is equally applicable to services connecting with external data sources.

***AC.1* User centric access control (✓)** Considering the user centrality that has been kept in mind during system design, the fulfillment of this requirement falls in line with the fulfillment of the non-function requirement of overall user centrality.

***AC.2* Fine-grained access control (✓)** The permission model provides users with fine-grained control over access to their data. This means that users always have complete control over who can access their data.

***AO.1* Opt-ins for data usage per application (✓)** Through service consumption permissions applications require explicit opt-in before data is available.

***AO.2* Opt-ins for different purposes of data usage per application (✓)** Because applications can specify multiple service consumptions for different purposes, users are equally capable of accepting or rejecting access for different purposes.

***AO.3* Revoking opt-ins at any time (✓)** Since every request requires explicit permission revoking permissions naturally fits within the framework. This could become more challenging when caches are added for increasing platform throughput. In this situation cache invalidation becomes a challenge. Nevertheless revokal of permissions is a key part of the model.

***AO.4* Configuring scope of data usage (✓)** The scope of data usage can be adjusted through permission rules. As the scalability tests indicate the number of permissions rules does have an effect on performance. However, as outlined in subsection 6.3.10, the number of permission rules that need to be evaluated for every request can be reduced through limitations and additional relationship management.

***PR.1* Viewing data usage of applications after opt-ins have been granted (✗)** Currently the platform does not implement additional logging. However, since the Central API handles every request logging of request meta data can be implemented trivially. These logs can subsequently be processed and made available such that users can inspect what applications do with their data.

***PR.1* Authentication via existing infrastructure (✓)** The proof of concept implementation uses oAuth2.0 for identity confirmations. Considering the oAuth2.0 capabilities of IBM Connections this is a natural fit.

***PR.2* Extending available data services with application level data services (✓)** Applications can expose additional data services be registering node types and services. Extensibility thus is a core part of the application model.

***PR.3* Platform provided privacy management (✓)** Through the Central API application developers can make use of existing privacy management infrastructure without needing to reinvent the wheel.

***PR.4* Infrastructure for gradually upgrading various platform components and providing dependent applications with sufficient means to handle version bumps (✓)** Similarly to the requirement of dealing with policy changes (item *PP.4*) this requirement is fulfilled through a versioning system at the core of the application model.

***PR.5* Receiving real time data events within the system (✗)** Currently the platform does not implement real time distribution of events within the system due to time constraints. However, initial attempts have been made to implement such functionality and results were promising. Due to the application model describing exact data needs for applications it is trivial to determine what applications have access to what data when new data is created. This knowledge can subsequently be used to push graph model changes to interested applications. However, creating a scalable implementation of this functionality requires additional work in terms of efficient data distribution and distribution protocols. At a minimum an additional core component can be created that will be invoked after the graph model is updated. This component will determine which applications have permission to access the changed data and notify these applications accordingly. The applications would then have to retrieve the data through the Central API.

6.4.3 Privacy principles

The privacy principles given in section 3.1 outline various practices that should be used by systems that handle personal information. Since the platform itself is also a system that handles personal information these principles are shortly discussed.

PRIVACY.1 **Collection limitation principle** The platform collects a lot of meta data about users. The effort has been made to store as little data as possible while still providing enough information to determine access rights. This is done by collecting only meta data about users. Even though this meta data is collected for the purpose of access control it does provide some privacy concerns. In order to use the collection limitation principle users of the platform should therefore sufficiently be informed of the role of the platform and the privacy implications.

PRIVACY.2 **Data Quality Principle** Several measures have been taken to ensure data quality, namely the use of replicated fault-tolerant persistent storage and eventual consistency of the graph model.

PRIVACY.3 **Purpose Specification Principle** The purpose of data collection within the system is clear; providing users with elaborate privacy management. However, additional work is required to inform users of this purpose.

PRIVACY.4 **Use Limitation Principle** The data stored within the graph model is solely used for access control purposes.

PRIVACY.5 **Security Safeguards Principle** This principle falls in line with the non-functional requirement that the system should be secure. Given the efforts made towards this goal this principle is partly satisfied. Nevertheless additional work is required to ensure system security.

PRIVACY.6 **Openness Principle** The current implementation provides user with full insights into what data is stored about them through services that expose the ownership within the graph model.

PRIVACY.7 **Individual Participation Principle** In line with the previous principle efforts have been made to construct services that expose the ownership within the graph model to users. Additional efforts could be made to ensure that users are in full control over data they own. For instance, it could be made mandatory for applications to let owners of data delete the data from their system.

PRIVACY.8 **Accountability Principle** The accountability principle becomes more important when the proof of concept is deployed in a real environment. Therefore, this principle should be kept in mind during such an event.

Chapter 7

Conclusions

In an increasingly connected work environment, data plays an important role. In this work some of the technical challenges that are paired with the increased use of personal data by enterprise-class applications have been tackled. Through a literature survey on the concepts of authentication, privacy policies, access control models, provenance and authorization several key observations are made. By combining these observations with the requirements from an real-world multinational enterprise, a simple data model has been constructed that describes privacy aware data sharing between different applications. Using this simple model an implementation was made that has scalable technologies as a foundation.

Testing of the proof of concept implementation shows signs that the model can be successfully be deployed to a scalable architecture. Although performance needs to be improved before large scale deployment is possible, several optimizations points remain open. Therefore an initial foundation is made for a privacy aware infrastructure for personalized enterprise applications.

7.1 Future work

The model as demonstrated in this work provides a basis for a privacy aware data sharing platform for enterprise applications. Although some of the challenges for such a platform have been tackled there are numerous challenges left.

7.1.1 Implementation optimizations

As mentioned in subsection 6.3.10 some additional efforts can be made to improve overall throughput. Depending on the outcome of these optimizations it might be necessary to introduce additional restrictions to the model in order to reduce the overhead.

7.1.2 Solution sustainability

Although there is room for improvement in terms of platform performance another question that arises is whether the solution is actually sustainable. Sustainability is important in both technical aspects as well as in usage aspects. The technical aspects include things such as the ability of the system to stay consistent under heavy usage. Additionally, management of the graph model must be sustainable. This could especially become challenging when third party applications make more extensive use of the relational side of the graph model by introducing custom relation types. Note that these sustainability challenges are inherit to the data model, not necessarily the implementation.

For the usage aspects things such as usability are of importance. When every application requires explicit opt-ins for every API call users might become desensitized. These users might accept or reject every opt-in without reading the privacy implications. In such a scenario the solution might not be sustainable because it is too fine grained.

7.1.3 Designing privacy management infrastructure

Closely linked with the previous point future work could focus on the management side of the platform. A danger of a fine-grained privacy management system is that users might become annoyed by the abundance of opt-ins that have to be managed. In future work an efficient management infrastructure can be designed to combat these issues. Such an infrastructure could include a privacy management overview that allows for easily managing opt-ins for various applications. For this overview it should be possible to set opt-ins on a fine-grained level such as what is possible in the underlying data model. Additionally the overview can provide functionalities such as default settings or quick overviews of data usage per application. Besides a management overview the infrastructure could also refine the versioning model such that it is more accessible and manageable by users.

7.1.4 Dealing with sparse data

In this work a lot of focus has been put on the user side of the problem. For this side, the data model clearly determines who has access to certain data and who does not. However, little focus was put on the implications of this model on application development. When application developers use the platform it could be possible that a lot of data is not available due to missing opt-ins. This requires application developers to be prepared to deal with sparse data. Furthermore, additional functionality could be needed to make dealing with sparse data easier. Here questions arise such as how application developers can know what data is available to a certain user.

7.1.5 Integration of the models in development environments

Currently application meta data needs to be constructed manually through either JSON files or by using the Application management client as shown in Appendix B. In either case the administration is rather cumbersome, especially when dealing with a lot of services. This could lead to situations in which the implemented services and meta data stored in the Applications repository are out of sync.

During development of the core components a possible solution for a potentially more sustainable way of generating application meta data was conceived. This solution is inspired by the

OpenAPI Specification[1]. This specification can be used to describe REST APIs. This is similar to what the Application repository does, although here more restrictions are applied to how the REST APIs are structured. For generating an OpenAPI Specification several tools are available for various languages and frameworks including JAX-RS. For JAX-RS additional annotations are available that can be used in conjunction with existing JAX-RS annotations to provide additional meta data about services. These annotations are then used to generate a JSON description of the OpenAPI specification. A similar approach could be useful for the application model. This way, maintaining services might become less cumbersome.

The following code segment shows an example of how additional annotations could be used to describe a service within the application model:

```
@ApplicationResource
public interface UsersResource implements Resource {

  @Service(
    edgeNodeType = User.class,
    httpMethod = HttpMethod.POST,
    version = 1L)
  @Consumes(MediaType.APPLICATION_JSON)
  @Produces(MediaType.APPLICATION_JSON)
  User createUser(
    User user);

  @Service(
    targetNodeType = User.class,
    httpMethod = HttpMethod.GET,
    version = 1L)
  @Produces(MediaType.APPLICATION_JSON)
  User getUser(
    @PathParam(ID) String userId);

  @Service(
    targetNodeType = User.class,
    edgeNodeType = Friendship.class,
    httpMethod = HttpMethod.GET,
    version = 1L)
  @Produces(MediaType.APPLICATION_JSON)
  Collection<Friendship> getFriends(
    @PathParam(ID) String userId);
}
```

[1] https://github.com/OAI/OpenAPI-Specification

7.1.6 Enforcing policy compliance

Currently trust plays an important role in the privacy model. Application developers are expected to be honest about their data usage. This means that every application should act in compliance with accepted service policies. However, due to the inherit loss of control when data is provided to external applications situations might occur in which compliance is not guaranteed. Several approaches can be used to tackle this problem, thus making it a potential subject for future work. Example approaches include introducing approval processes for applications making use of the platform. Alternatively infrastructure can be set up for users to alert system administrators of malicious data usage.

Bibliography

[1] Alessandro Acquisti. The economics of personal data and the economics of privacy. 2010.

[2] Claudio Agostino Ardagna, Marco Cremonini, S De Capitani di Vimercati, and Pierangela Samarati. A privacy-aware access control system. *Journal of Computer Security*, 16(4):369–397, 2008.

[3] Claudio Agostino Ardagna, Ernesto Damiani, S De Capitani di Vimercati, and Pierangela Samarati. Towards privacy-enhanced authorization policies and languages. In *Data and applications security XIX*, pages 16–27. Springer, 2005.

[4] Claudio Agostino Ardagna, S De Capitani di Vimercati, and Pierangela Samarati. Enhancing user privacy through data handling policies. In *Data and Applications Security XX*, pages 224–236. Springer, 2006.

[5] Paul Ashley, Satoshi Hada, Günter Karjoth, and Matthias Schunter. E-p3p privacy policies and privacy authorization. In *Proceedings of the 2002 ACM workshop on Privacy in the Electronic Society*, pages 103–109. ACM, 2002.

[6] Michael Beiter, Marco Casassa Mont, Liqun Chen, and Siani Pearson. End-to-end policy based encryption techniques for multi-party data management. *Computer Standards & Interfaces*, 36(4):689–703, 2014.

[7] Tyrone Cadenhead, Vaibhav Khadilkar, Murat Kantarcioglu, and Bhavani Thuraisingham. A language for provenance access control. In *Proceedings of the first ACM conference on Data and application security and privacy*, pages 133–144. ACM, 2011.

[8] Rick Cattell. Scalable sql and nosql data stores. *ACM SIGMOD Record*, 39(4):12–27, 2011.

[9] Xiao Chen. Google big table. *2010*.

[10] Lorrie Cranor, Marc Langheinrich, Massimo Marchiori, Martin Presler-Marshall, and Joseph Reagle. The platform for privacy preferences 1.0 (p3p1. 0) specification. *W3C recommendation*, 16, 2002.

[11] Luca Galli and Piero Fraternali. Achievement systems explained. In *Trends and Applications of Serious Gaming and Social Media*, pages 25–50. Springer, 2014.

[12] Hans Peter Gassmann. Oecd guidelines governing the protection of privacy and transborder flows of personal data. *Computer Networks (1976)*, 5(2):127–141, 1981.

[13] Satoshi Hada and Hiroshi Maruyama. Session authentication protocol for web services. In *Applications and the Internet (SAINT) Workshops, 2002. Proceedings. 2002 Symposium on*, pages 158–165. IEEE, 2002.

[14] Dick Hardt. The oauth 2.0 authorization framework. 2012.

[15] KA Hargreaves and K Berry. Regex. *Free Software Foundation*, 675, 1992.

[16] Marc Hüffmeyer and Ulf Schreier. Efficient attribute based access control for restful services. In *ZEUS*, pages 55–62, 2015.

[17] Michael Jones, Paul Tarjan, Yaron Goland, Nat Sakimura, John Bradley, John Panzer, and Dirk Balfanz. Json web token (jwt). 2012.

[18] Sabrina Kirrane, Alessandra Mileo, and Stefan Decker. Access control and the resource description framework: A survey.

[19] Butler W Lampson. Protection. *ACM SIGOPS Operating Systems Review*, 8(1):18–24, 1974.

[20] Paul J Leach, John Franks, Ari Luotonen, Phillip M Hallam-Baker, Scott D Lawrence, Jeffery L Hostetler, and Lawrence C Stewart. Http authentication: Basic and digest access authentication. 1999.

[21] Jun Li, Bryan Stephenson, Hamid R Motahari-Nezhad, and Sharad Singhal. A data assurance policy specification and enforcement framework for outsourced services. *HP Laboratories*, 2009.

[22] Luc Moreau, Ben Clifford, Juliana Freire, Joe Futrelle, Yolanda Gil, Paul Groth, Natalia Kwasnikowska, Simon Miles, Paolo Missier, Jim Myers, et al. The open provenance model core specification (v1. 1). *Future Generation Computer Systems*, 27(6):743–756, 2011.

[23] Qun Ni, Shouhuai Xu, Elisa Bertino, Ravi Sandhu, and Weili Han. An access control language for a general provenance model. In *Secure Data Management*, pages 68–88. Springer, 2009.

[24] Jaehong Park, Dang Nguyen, and Ravi Sandhu. A provenance-based access control model. In *Privacy, Security and Trust (PST), 2012 Tenth Annual International Conference on*, pages 137–144. IEEE, 2012.

[25] Joon S Park, Gaeil An, and Ivy Y Liu. Active access control (aac) with fine-granularity and scalability. *Security and Communication Networks*, 4(10):1114–1129, 2011.

[26] Siani Pearson. Privacy management in global organisations. In *Communications and Multimedia Security*, pages 217–237. Springer, 2012.

[27] Santiago Pericas-Geertsen and Marek Potociar. Jax-rs: Java api for restful web services. *Oracle Corporation*, pages 1–84, 2013.

[28] Joseph Reagle and Lorrie Faith Cranor. The platform for privacy preferences. *Communications of the ACM*, 42(2):48–55, 1999.

[29] Pierangela Samarati and Sabrina De Capitani Di Vimercati. Access control: Policies, models, and mechanisms. *Lecture notes in computer science*, pages 137–196, 2001.

[30] Alberto Siena, Silvia Ingolfo, Anna Perini, Angelo Susi, and John Mylopoulos. Automated reasoning for regulatory compliance. In *Conceptual Modeling*, pages 47–60. Springer, 2013.

[31] Alberto Siena, Ivan Jureta, Silvia Ingolfo, Angelo Susi, Anna Perini, and John Mylopoulos. Capturing variability of law with nomos 2. In *Conceptual Modeling*, pages 383–396. Springer, 2012.

[32] Alberto Siena, John Mylopoulos, Anna Perini, and Angelo Susi. Designing law-compliant software requirements. In *Conceptual Modeling-ER 2009*, pages 472–486. Springer, 2009.

[33] Robert-Jan Sips, Alessandro Bozzon, Gerard Smit, and Geert-Jan Houben. The inclusive enterprise: Vision and roadmap. In *Engineering the Web in the Big Data Era*, pages 621–624. Springer, 2015.

[34] Haoyu Song, Sarang Dharmapurikar, Jonathan Turner, and John Lockwood. Fast hash table lookup using extended bloom filter: an aid to network processing. *ACM SIGCOMM Computer Communication Review*, 35(4):181–192, 2005.

[35] OASIS Standard. extensible access control markup language (xacml) version 2.0, 2005.

[36] Romuald Thion, François Lesueur, and Meriam Talbi. Tuple-based access control: a provenance-based information flow control for relational data. 2015.

Appendix A

Implementing an image sharing service

As an example of how the model constructed in this work can be used this appendix shows how a simple image sharing service API can be created. This implementation uses the same technology stack and conventions as core components. This application provides services for users to upload and view images. This example application implementation illustrates how the model can be used to describe a simple service and shows how the platform can benefit the simplicity of implementation.

A.1 Application definition (mainly for constants)

```
@ApplicationDescription
public final class ImagesApplicationDescription {
  @ApplicationId
  public static final String APPLICATION_ID = "images";

  private ImagesApplicationDescription() { }
}
```

A.2 NodeType definitions

```
@Table(
    keyspace = Image.NODE_TYPE_APPLICATION_ID,
    name = Image.NODE_TYPE_NAME)
@NodeModel
public class Image extends AbstractCassandraModel {

  @NodeTypeApplicationId
  public static final String NODE_TYPE_APPLICATION_ID =
      ImagesApplicationDescription.APPLICATION_ID;
```

```
@NodeTypeName
public static final String NODE_TYPE_NAME = "image";

@NodeTypeId
public static final String NODE_TYPE_ID =
    NODE_TYPE_APPLICATION_ID + $ + NODE_TYPE_NAME;

@NodeIdPattern
public static final String NODE_ID_PATTERN =
      "(" + User.NODE_ID_PATTERN + ")\\."
    + "(" +  UUID_PATTERN + ")";

@TargetNodePath
public static final String TARGET_NODE_PATH =
    NODE_TYPE_ID + ID_START + NODE_ID_PATTERN + ID_END;

@JsonView(Views.Public.class)
@PartitionKey(0)
@IdSegment(0)
@Column(name = "user_id")
private String userId = NULL;

@JsonView(Views.Public.class)
@PartitionKey(1)
@IdSegment(1)
@Column(name = "uuid")
private UUID uuid;

@JsonView(Views.Public.class)
@Column(name = "mime_type")
private String mimeType;

@JsonView(Views.Public.class)
@Column(name = "title")
private String title;

@JsonView(Views.Public.class)
@Column(name = "description")
private String description;

@JsonIgnore
@Column(name = "image_data")
```

```
  private ByteBuffer imageData;

  @JsonView(Views.Public.class)
  @Column(name = "created")
  private Date created;
}
```

A.3 REST service descriptions

```
@Path("")
public interface ImagesResource extends Resource {
  @POST
  @Path(Image.NODE_TYPE_ID)
  @Consumes(MediaType.MULTIPART_FORM_DATA)
  @Produces(MediaType.APPLICATION_JSON)
  @JsonView(Views.Private.class)
  Image createByAnonymous(
    IMultipartBody multipartBody);

  @POST
  @Path(User.TARGET_NODE_PATH + "/" + Image.NODE_TYPE_ID)
  @Consumes(MediaType.MULTIPART_FORM_DATA)
  @Produces(MediaType.APPLICATION_JSON)
  @JsonView(Views.Private.class)
  Image createByUser(
    @PathParam(ID) String userId,
    IMultipartBody multipartBody);

  @GET
  @Path(Image.TARGET_NODE_PATH)
  @Produces(MediaType.APPLICATION_JSON)
  @JsonView(Views.Public.class)
  Image get(@PathParam(ID) String id);

  @GET
  @Path(Image.TARGET_NODE_PATH)
  @Produces("image/*")
  Response getRaw(@PathParam(ID) String id);

  @DELETE
  @Path(Image.TARGET_NODE_PATH)
  @Produces(MediaType.APPLICATION_JSON)
  @JsonView(Views.Private.class)
  Image delete(@PathParam(ID) String id);
}
```

A.4 JAX-RS endpoint used in a web.xml in combination with the servlet 3.1 spec

```
public class ImagesEndpoint extends AbstractEndpoint {

  @Override
  protected Stream<Class<?>> getResources() {
    return Stream.of(
      ImagesResource.class);
  }

  @Override
  protected Stream<Class<?>> getProviders() {
    return Stream.concat(
      super.getProviders(),
      Stream.of(PostLocationFilter.class));
  }
}
```

A.5 Deployement and platform benefits

Finally the REST interface should be implemented using some storage technology. After this the code can be deployed to a WebSphere Liberty application server and should only be able to communicate with the central API. By structuring applications in this way a lot of functionality is offered by the platform. By properly setting up service descriptions using the applications manager client the image sharing API can be used by users to upload images and share them only with those who they approved of. An example of this process will be given in Appendix B.

Appendix B

Managing applications

The applications repository plays an important role in the designed platform. In this repository all information is contained about how applications interact with data. During proxy request handling this repository is used to determine validity of requests. For this reason it is important that the application repository provides an accurate picture of reality. To facilitate the management of applications a simple application management client application has been created. In line with the example image uploading service the process of managing service is used as an example of how this management application can be used.

The first step is creating the application, shown in Figure B.1.

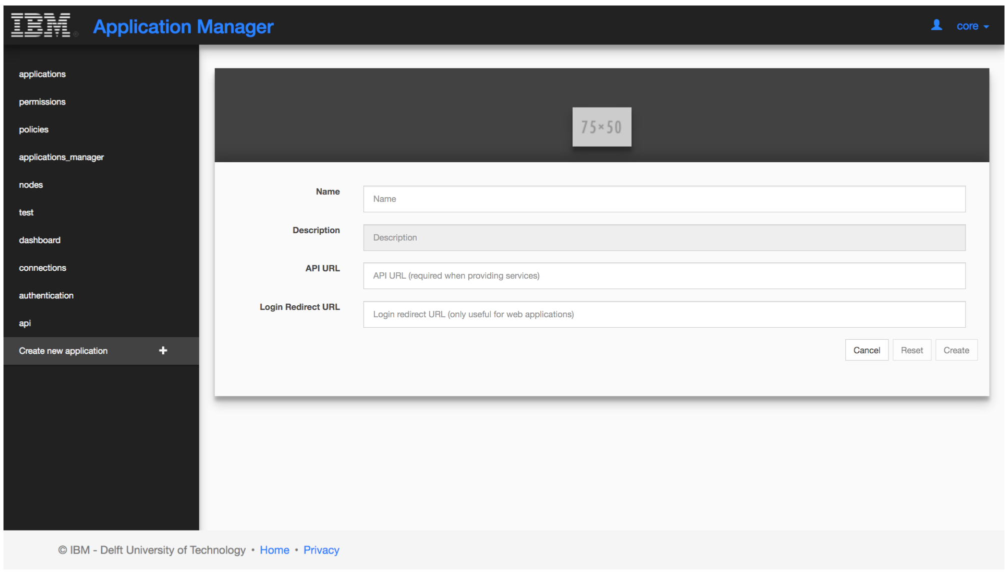
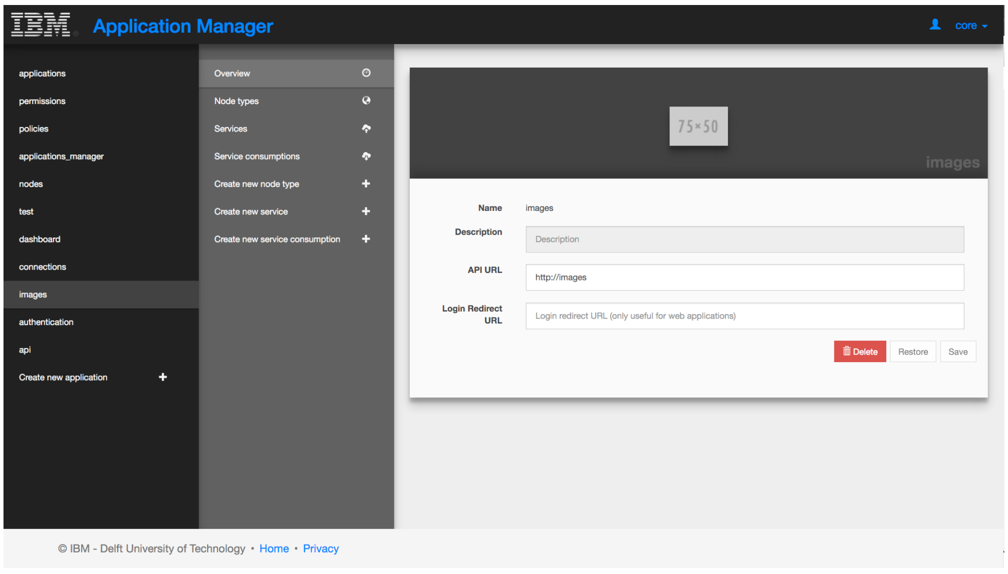


Figure B.1: Creating the application

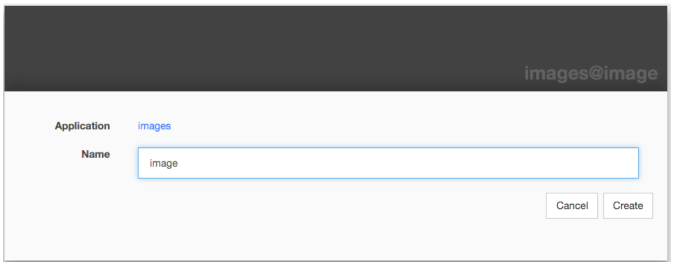
After an application has been created it appears in the overview, shown in Figure B.2. This overview can be used to configure more details about the application.



Name	images
Description	Description
API URL	http://images
Login Redirect URL	Login redirect URL (only useful for web applications)

Figure B.2: Created application

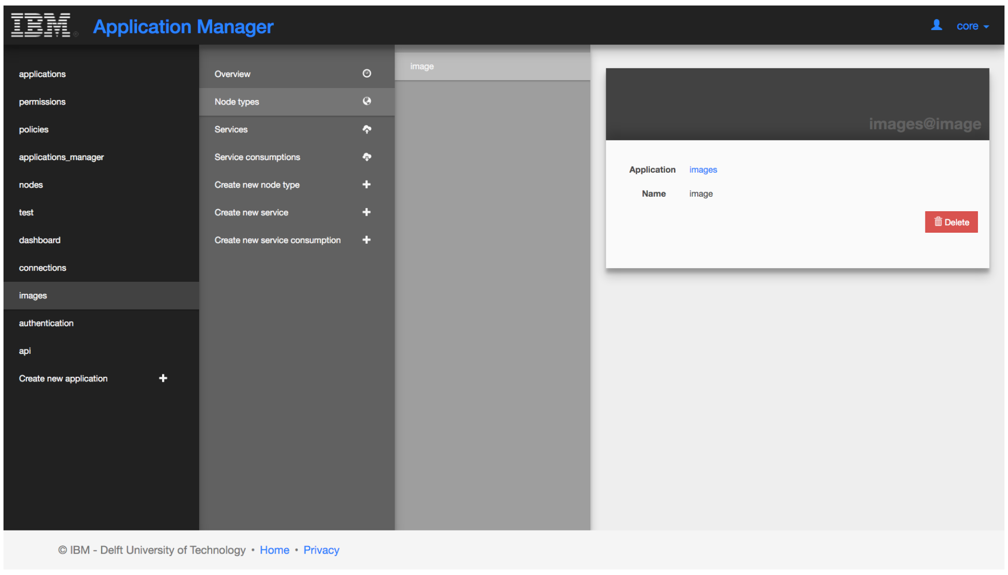
Now that the application is available a new node type can be created, shown in Figure B.3. When creating a node type only a name is required. For this example an image node type can be created, which is the main data type of the images application.



Application	images
Name	image

Figure B.3: Creating the image node type

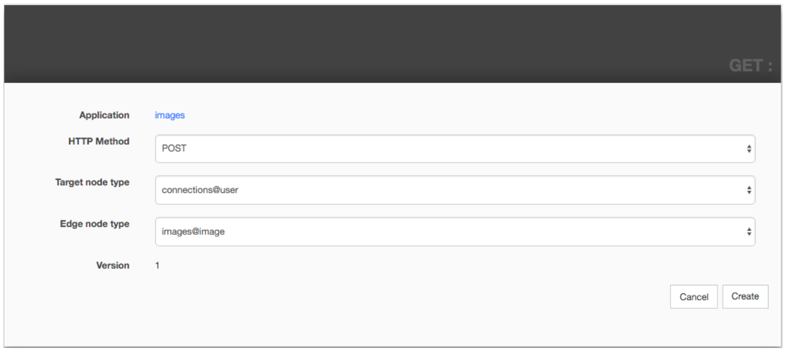
After a node type has been created it will appear in the overview, shown in Figure B.4. Currently this overview provides little information, but this overview might include additional information such that application developers can more easily determine what data is available.



Application images
Name image

Figure B.4: Created node type

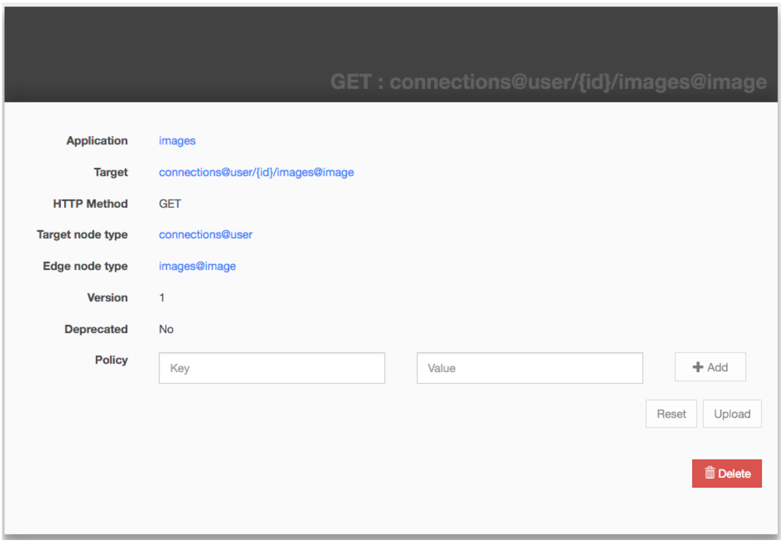
Now that a node type is available it is possible to create a new service, shown in Figure B.5. This figure shows the different properties that can be adjusted for services. In this case a new service is created that represents the *createByUser* method from section A.3. This service can be used to create an image that is owned by a user.



Application images
HTTP Method POST
Target node type connections@user
Edge node type images@image
Version 1

Figure B.5: Creating a service

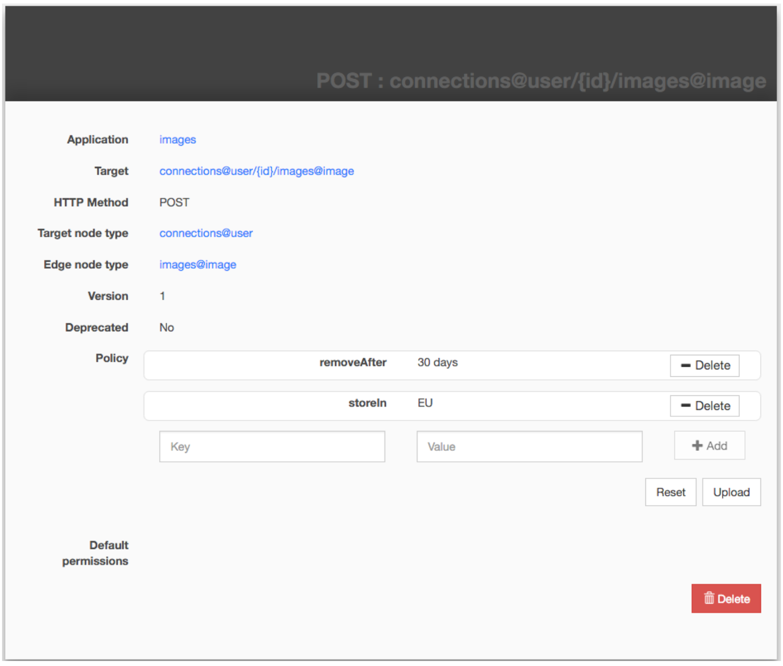
After the service has been created an overview of the service is available, shown in Figure B.6. Note that for this service no policy exists yet.



Application	images
Target	connections@user/{id}/images@image
HTTP Method	GET
Target node type	connections@user
Edge node type	images@image
Version	1
Deprecated	No
Policy	

Figure B.6: Created service

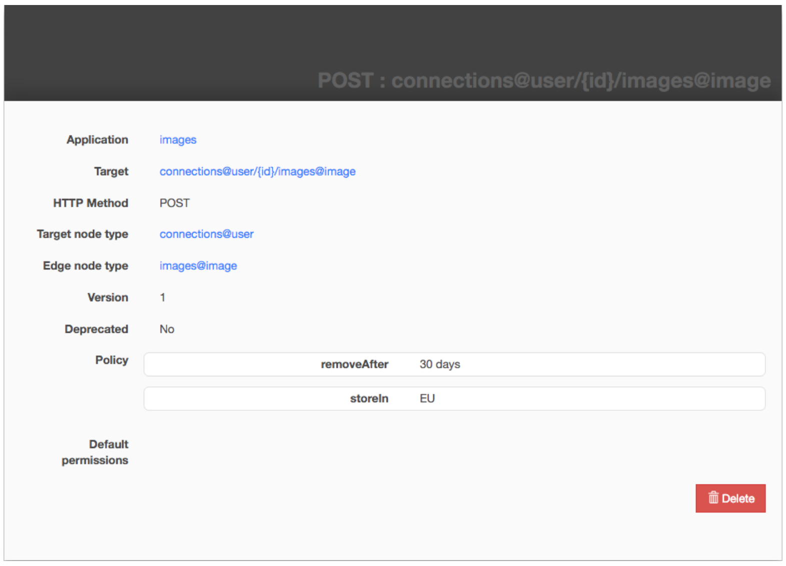
Before the service is usable a policy should be provided, the process of which is shown in Figure B.7. As a proof of concept a policy simply consists of key value pairs, although a more elaborate policy specification language should be included here in the future.


POST : connections@user/{id}/images@image

Application	images
Target	connections@user/{id}/images@image
HTTP Method	POST
Target node type	connections@user
Edge node type	images@image
Version	1
Deprecated	No
Policy	removeAfter: 30 days; storeIn: EU
Default permissions	

Figure B.7: Uploading a policy

After the policy has been uploaded it can not be changed anymore, shown in Figure B.8. Policy changes would require a new version of the service. A similar process would be required for default permissions.



POST : connections@user/{id}/images@image

Application	images
Target	connections@user/{id}/images@image
HTTP Method	POST
Target node type	connections@user
Edge node type	images@image
Version	1
Deprecated	No
Policy	removeAfter 30 days
	storeIn EU
Default permissions	

Figure B.8: Uploaded policy

B.1 Conclusions

The current version of the applications manager client provides basic utilities for managing applications. The process outlined above shows the steps required when deploying services to the platform. Although a basis for this client is built there are a lot of additional features that would be useful in a real world environment. For instance, the application overview would be much more usable of it was possible to inspect services of other applications together with detailed information about the usage. Such a client could also be combined with existing documentation technologies such as the previously mentioned OpenAPI Specification. By providing such resources through a single application it would be easy for developers to search for relevant functionalities among the available services.