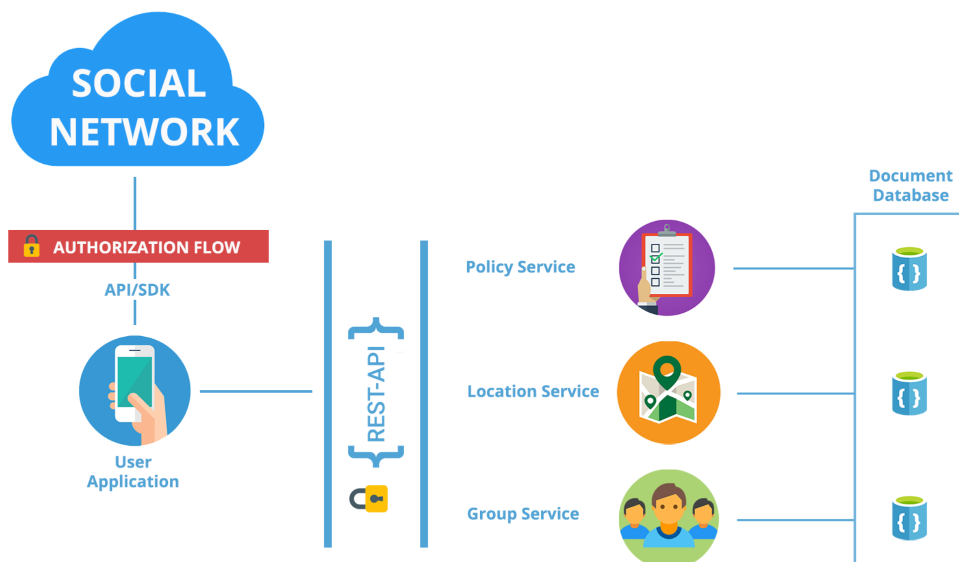



# Dynamic Privacy Policies with Multiple Contexts

Master's thesis in Computer science and engineering

ARBNOR BILJALI
VALON HUSKAJ

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2021

Master's thesis 2021

# Dynamic Privacy Policies with Multiple Contexts

ARBNOR BILJALI
VALON HUSKAJ

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden 2021

© ARBNOR BILJALI
VALON HUSKAJ, 2021.

Supervisor: Gerardo Schneider, Department of Computer Science and Engineering
Advisor: Raul Pardo, IT University of Copenhagen
Examiner: Andrei Sabelfeld, Department of Computer Science and Engineering

Master's Thesis 2021
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Gothenburg, Sweden 2021

Arbnor Biljali
Valon Huskaj
Department of Computer Science and Engineering
Chalmers University of Technology

# Abstract

Privacy polices are widely used on Social Network Services(SNS) in order to limit the audience on certain information posted on the SNS. In this master thesis we are concerned with the problem of defining dynamic privacy policies with regards to multiple contexts such as time and location on multiple SNS simultaneously. For that we have designed and implemented an application to evaluate the thesis work. Our results show that it is possible to use a centralized client application to send content towards multiple SNS simultaneously. This due to the design of an architecture that allows multiple integration of SNS on the same application. Another result of the contributing architecture is the support of one, two or more contexts in the same dynamic privacy policy. This is possible because of the contextual information, such as GPS location etc, derived from the client device itself.

Keywords: Privacy policy, Dynamic privacy policy, Social Network, Conditional policies, Contextual information.

# Acknowledgements

We would like to thank our Chalmers supervisor Gerardo Schneider for valuable meetings and input. We would also like to thank Raúl Pardo at IT University of Copenhagen for providing useful thoughts and valuable input on the master thesis.

Arbnor Biljali, Valon Huskaj, Gothenburg, February 2021

# Contents

**List of Figures** **xi**

**1 Introduction** **1**
1.1 Problem Description . . . . . . . . . . . . . . . . . . . . . . . . . . 1
1.2 Goals . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2
1.3 Limitations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
1.4 Methodology . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4
1.5 Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4

**2 Background** **5**
2.1 Privacy Policies . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
2.1.1 Static Privacy Policy . . . . . . . . . . . . . . . . . . . . . . 5
2.1.2 Dynamic Privacy Policy . . . . . . . . . . . . . . . . . . . . 5
2.2 Building Block Method . . . . . . . . . . . . . . . . . . . . . . . . . 6
2.3 Microservice Architecture . . . . . . . . . . . . . . . . . . . . . . . 6
2.4 RESTful API . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
2.5 Authorization Flow . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
2.6 Client Server Architecture . . . . . . . . . . . . . . . . . . . . . . . 8
2.7 Diaspora . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9

**3 Architecture & Design** **10**
3.1 Dynamic Privacy Policy Framework . . . . . . . . . . . . . . . . . . 10
3.1.1 Policy Creation . . . . . . . . . . . . . . . . . . . . . . . . . 10
3.1.2 Policy Evaluation . . . . . . . . . . . . . . . . . . . . . . . . 11
3.2 System Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12

**4 Implementation** **14**
4.1 Server Side Implementation . . . . . . . . . . . . . . . . . . . . . . 14
4.1.1 Policy Service . . . . . . . . . . . . . . . . . . . . . . . . . . 14
4.1.2 Group Service . . . . . . . . . . . . . . . . . . . . . . . . . . 18
4.1.3 Location Service . . . . . . . . . . . . . . . . . . . . . . . . 20
4.2 Client Side Implementation . . . . . . . . . . . . . . . . . . . . . . 21
4.2.1 User Application . . . . . . . . . . . . . . . . . . . . . . . . 21
4.2.2 Server Side Integration . . . . . . . . . . . . . . . . . . . . . 22
4.2.3 Authorization Flow . . . . . . . . . . . . . . . . . . . . . . . 22
4.2.3.1 Diaspora . . . . . . . . . . . . . . . . . . . . . . . 22
4.2.3.2 Facebook . . . . . . . . . . . . . . . . . . . . . . . 25

4.2.4 Social Network Integration . . . 27
4.2.4.1 Diaspora API . . . 28
4.2.4.2 Facebook API and SDK . . . 28
4.3 Algorithm . . . 29
4.3.1 Evaluation Algorithm . . . 29
4.3.1.1 Evaluation Methods . . . 30
4.3.2 Policy Priority Algorithm . . . 31

**5 Results** **32**
5.1 Predefined scenario . . . 32
5.2 Multiple Social Networks . . . 33
5.3 Multiple Context . . . 35
5.4 Securing Contextual Information . . . 38

**6 Discussion** **39**
6.1 Conclusion . . . 40
6.2 Future Work . . . 41

**Bibliography** **42**

# List of Figures


2.1 A graphical view of monolith vs microservices [15] . . . . . . . . . . . 6
2.2 OAuth2 Generic Flow . . . . . . . . . . . . . . . . . . . . . . . . . 8

3.1 Visual presentation of the segments in Policy 3 . . . . . . . . . . . . . 11
3.2 Overview of the System . . . . . . . . . . . . . . . . . . . . . . . . 12

4.1 An overview of the integration towards the different services. . . . . . 22
4.2 Diaspora Permission Dialog . . . . . . . . . . . . . . . . . . . . . . 24
4.3 Facebook for Developer . . . . . . . . . . . . . . . . . . . . . . . . . 25
4.4 Facebook Login Dialog . . . . . . . . . . . . . . . . . . . . . . . . . 26
4.5 Overview of the Social Network Integration . . . . . . . . . . . . . . . 27

5.1 Triggering an event . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
5.2 Posting to multiple SNS . . . . . . . . . . . . . . . . . . . . . . . . 34
5.3 The post represented in each SNS. . . . . . . . . . . . . . . . . . . . . 35
5.4 Choose event type and groups for which the policy should apply . . . 35
5.5 Choose variables for Time context . . . . . . . . . . . . . . . . . . . 36
5.6 Choose variables for Location context . . . . . . . . . . . . . . . . . 36
5.7 Naming the policy and viewing the readable string . . . . . . . . . . 37
5.8 Bob's and David's Stream with Alice posts after all scenarios . . . . . 38

# 1

# Introduction

The beginning of the 21st century has brought the social aspect of living to the internet through services such as Facebook (2004), LinkedIn (2002) and Twitter (2006), called Social Network Services (SNS). These services allow each individual to communicate, share information and be part of a digital community.

Sharing data on an SNS has become fairly easy and the amount of data that is shared increases daily. Poorly handled privacy management has caused users to share more information than intended [1]. This has lead to political involvement which in turn has resulted to the largest privacy scandal in modern history [2].
One of many reasons for this problem is that all well known SNS only provide static privacy policies that are applied on events posted by the user. For example on Facebook, a user could choose the following policy:

**Policy 1** *Who can see my upcoming photos?*

with the limited choices of Public, Friends, and Friends of friends. The policy itself has no variables that can change, all photos uploaded will have static policy 1 applied. This means that the user can not create policies based on changing variables, such as time or location. Policies with changing variables are known as *dynamic privacy policies* (DPP).

This thesis defines DPP as conditional policies that hold or not depending on temporal and spacial conditions, referred to as *context*. There exists no current solution where the user can define a DPP and use the policy on multiple SNS at the same time.

## 1.1 Problem Description

By having a static policy, the user is limited to a set of predefined rules by the SNS. Furthermore, these predefined rules can not be changed to allow varying context properties. A scenario where we can see the limitation of static policies would be for e.g. when a user has content that differs between weekdays and weekends. Meaning, on weekend the user is politically active and does not want any colleagues to see the involvement. However, during weekdays the user shares several work related content and is happy to share this with the colleagues. This requires the user for each post to exclude or include a set of connected people. With a static policy, there is no

flexibility to adapt the content to the audience based on a context, such as time in this example.

Using DPP is a solution to this limitation. By using a DPP framework, the user would be able to easily create a policy where colleagues are forbidden to see the users content during weekend, this due to the support of varying context properties. However, the development of such a framework is difficult and complex.

The first challenge is to create a logical expression of the DPP that will support several contexts. Several contexts should also be able to be enforced on the same policy, e.g. a user who does not want colleagues to see her content during weekends and when the user is at home. A policy that needs to consider both time and location during the same event.

Another challenge lies in allowing this logical expression to be used in a user-friendly syntax, meaning the user should be able create and read a complex policy with minimal effort.

Lastly, the framework needs to handle already defined policies on the SNS. If a policy already exists on the SNS e.g. that *allows* content to a certain group, and the framework has a policy that would *block* the same content to the same group then the framework needs to be able to handle such contradictions.

In Pardo et al. [3] the absence of DPP in today's SNS are brought up. Both the possibility of defining and enforcing DPP. As mentioned before the only set of policies available on SNS are the predefined static policies. Pardo et al. present a theoretical solution for defining and enforcing DPP along with a proof-of-concept implementation towards the SNS Diaspora [19].

In this thesis we propose an extension where we make use of the contextual information retrieved from the client device, such as timestamp, GPS location etc. This allows for defining more complex policies such as *allowing my friends to view my content when I'm at work*, where *work* refers to a specific GPS location. Another extensions which differs from Pardo et al. is the possibility of defining DPP that are enforced on multiple SNS simultaneously.

## 1.2 Goals

The goal in this thesis is divided into three parts, with each part being essential in creating a DPP Framework Architecture.

*How do we centralize DPP management towards multiple SNS*

The first part is to research how to design an architecture that allows the users, via our implemented solution, to manage one or more policies towards multiple SNS. The solution needs to be flexible in terms of extension, meaning new SNS can be

added without any big effort.

*How can we make use of multiple contexts*

The second part is to research how the architecture can implement the usage of multiple context on the same policy. How do we create an implementation that allows, e.g. both time and location, to be applied on a single policy.

*How do we secure the users contextual information*

And the last part is to review the proposed architecture and inspect how secure the solution is in terms of user privacy. This is important since our solution will have contextual information retrieved from the client device, therefore we need to ensure that this data is secured within the implemented solution.

Overall, our goal can be summarized into designing a secure architecture for managing DPP with the support for multiple contexts on a policy that can be used towards different SNS. Furthermore we need to implement the designed architecture in order to properly evaluate the feasibility of the design.

## 1.3 Limitations

A natural development of this thesis would be to consider two additional extensions. One extension is having previously added content change audience groups when already defined policies are updated or when new policies are created. This since users would assume changes to be propagate on all previous posts.
Second is ensuring privacy towards non-direct posts, meaning privacy towards e.g. friends of friends. This is useful since a lot of content shared include information about individuals with no direct relationship to the audience.
However these extensions will not be taken into consideration in this thesis and will be left out due to the reasons explained below.

**Policy changes with regards to previously created content.**

Most well known SNS will provide an integration that can be used to create content towards the SNS e.g. integration for uploading a photo on the SNS. These integrations may be limited by the SNS due to security or privacy reason. As for now, a known limitation for any SNS is that there exist no way to modify previously added content with a new policy. As an example, if at this moment we create a policy via the framework that should block a certain group of people to see content between 08:00-16:00. Then all new content created would be affected by this policy, however all content created prior to this moment would not be affected by this policy due to the limitations of the SNS.

**Privacy in non-direct posts. e.g. privacy towards friends of friends.**

If we consider an example where three individuals Alice, Bob, and Charlie use an SNS. Both Alice and Charlie are friends with Bob, but not with each other. Now if Alice has a privacy policy which states:

*Location should only be seen by my friends*

then this will mean that only Bob should be able to see where Alice is at any given time. But if Bob were to tag Alice on an picture on an location, then Charlie will see where Alice is. By doing this, Alice's policy is bypassed.
In order to fix this issue each friend needs to have access to each others privacy policies and act accordingly when posting. This is a project in and of it self since the policy information provided by the SNS are very limited. Therefore this will not be a part of this thesis.

## 1.4 Methodology

In this thesis we choose to make use of research and evaluation method. We research well known literature with context on the definition and creation of DPP. Later, we research some well know SNS and inspect the possibilities of retrieving data from these SNS. With the obtained knowledge we then design a DPP framework architecture that can be implemented with SNS. With the given design we then develop a proof of concept (POC) framework with integration towards two different SNS. Lastly, we evaluate the proposed architecture with our POC on the two integrated SNS. This will be done by using predefined scenarios with different users and verify if the content is visible based on the DPP.

## 1.5 Overview

The roadmap of this thesis is as follows: Chapter 2 gives a theoretical background about policies and the different techniques and concepts. In Chapter 3 we define the DPP framework and how it should handle a DPP and we also give an overview over the system. In Chapter 4 we take a look at the actual implementation of the architecture with details on all building-blocks as well as the algorithm used for evaluating policies. Chapter 5 presents the result derived from validating our solution. Lastly in Chapter 6 we discuss and come to conclusion towards the goals set on this thesis.

# 2
# Background

As described in the previous chapter, a static privacy policy is limited in terms of support for changing variables. This however can be solved by using a DPP, described in the same chapter. In this chapter we present the technologies that are needed to create a DPP framework that can be integrated with a existing SNS.

## 2.1 Privacy Policies

Privacy policies state what parties are allowed to view or take part of certain information. An example when privacy policies are in action could be when a user posts a photo on their board on an SNS. They user has a chosen audience group which might be friends, family or any. The privacy policy will then limit the information to that specific audience group.

### 2.1.1 Static Privacy Policy

When referring to a privacy policy as static privacy policy, it simply means that the user chooses an audience group for a certain event and when the users posts this event, the specific privacy policy will be used. This means that if the user, for instance chooses *Friends* as the audience group for the policy *Who can see my upcoming photos*, then all the photos posted will have *Friends* as audience group. The user still has the option to change the audience group for the event, this however has to be done manually for each event.

### 2.1.2 Dynamic Privacy Policy

With DPP the user can create a privacy policy that will change according to contextual information. For instance if a user has the policy:

**Policy 2** I don't want my colleagues to see my posts when I'm outside of work

then when the user posts outside of work the post will not have colleagues as an audience group. This in turn means that if the user posts again when at work then the colleagues will be a part of the audience group. Important to note is that the user doesn't need change the audience group on each event, this is done automatically according to the DPP.

## 2.2 Building Block Method

The Building Block Method (BBM) is a method for writing software system build for easy evolution and extension if needed [4]. The main concept is based on that you shall be able to add new features with minimal modifications on existing code. You will also try to achieve a reusable component architecture where components are isolated but reusable to create new features. As comparison, if we look at a multi-tool, such as a screwdriver with detachable heads. You only need one of the base, but you most likely have several heads that can be attached to solve a certain task or be combined to solve a mix of different tasks.

## 2.3 Microservice Architecture

Micro-service architecture is defined as an application where the services are independent from one another in terms of maintainability, testing and development. Unlike a monolithic architectural setup, one can easily make changes to one service without the need for downtime on other services in the application.
Redhat [15] defines Microservice as

*"Microservices are an architectural approach to building applications. As an architectural framework, microservices are distributed and loosely coupled, so one team's changes won't break the entire app."*

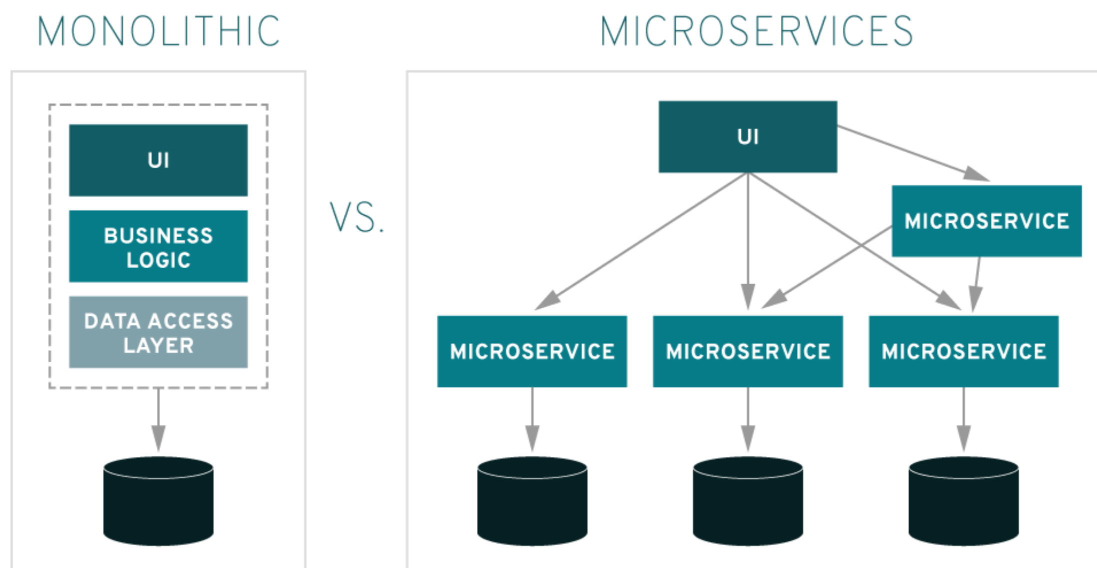



**Figure 2.1:** A graphical view of monolith vs microservices [15]

When compared to a monolithic architecture, the setup of an application which uses the microservice approach will normally consist of different services with exposed APIs or other communication methods which either a UI or another service can call. This can be seen in Figure 2.1.

Each service in the application will have their own database so that a failure in the database of one service doesn't brake the other services in the application.

## 2.4 RESTful API

REST is a useful software architectural style to use for allowing communication towards an application. **REST** stands for ***RE****presentational* ***S****tate* ***T****ransfer* and defines the different communication endpoints for a service or application.
The usage of this style when implementing an API, is typically called RESTful API. When making use of HTTP, which is the most common, the HTTP methods are available (such as POST,GET,DELETE, etc).
However, there are 6 constraints that make up a RESTful service or application. [16]

- Client-Server Architecture - *The server and client should not have a strong dependency between each other. They should be able to evolve separately and the client should only know the URIs.*
- Statelessness - *The server should not store any information about any historical request/data.*
- Cache-ability - *Resources that can be cached, should be declared as cache-able.*
- Layered System - *A client cannot normally tell where it is connected. On the last server or somewhere in between.*
- Uniform Interface - *Each resource is unique within the collaborative URI.*
- (Optional) Code on Demand - *Allow for code to be returned for a request. However, normally only static responses such as HTML,XML and JSON are being returned.*

If a service/application fulfill some of the constraints then that may be called a RESTful service/application, but in order to be called a *fully* RESTful service/application then all 6 constraints need to be fulfilled.

## 2.5 Authorization Flow

Implementing a secured solution for resource sharing in a software development project can be very difficult and challenging and even more challenging when accessing data between different services. Because of this a industry-standard protocol known as Oauth 2.0 has been developed to solve this type of problem. Many frameworks build upon the OAuth 2.0 standard exist today, such as Facebook Login [5], OpenID Connect [8], Sign in with Twitter [9] and many more.

In general the protocol allows a smooth integration between an Application (Client), Resource Owner, Authorization Server and Resource Server. Typically on an SNS scenario, a developer will register an Application on the SNS, e.g on Facebook this would be "Facebook for Developer". This application is created with a set of permissions that will allow requests and fetch data from the particular SNS, this can be permission to access your name, age and email. With a given authorization flow,

users can allow applications access to the requested permissions and start sharing its resources.

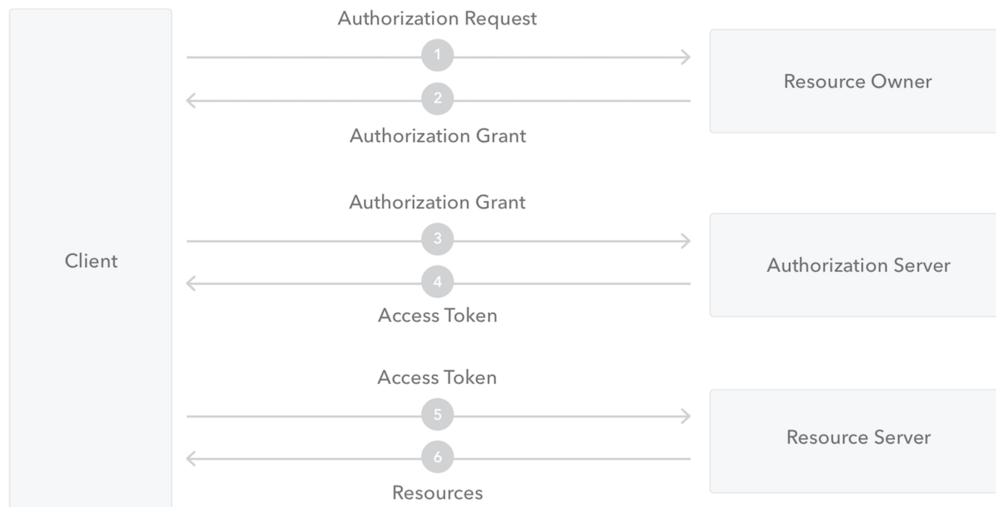



**Figure 2.2:** OAuth2 Generic Flow

In Figure 2.2 a generic flow is presented and it is implemented as follow:

1. The application makes a request for authorization (Authorization Request). Based on the provider, a list of permissions are presented and the resource owner is asked to allow or deny the request.
2. If the request is allowed (Authorization Grant), a Grant Token is sent back to the application.
3. With the Grant Token (Authorization Grant), a new request will be made to the Authorization Server to verify user and permissions on the selected provider.
4. If Grant Token is valid and user and permissions are correct, an Access Token is returned.
5. Access Token can then be used to make a request to the Resource Server and fetch data within the requested permission scope of the application.
6. If Access Token is valid, requested data (Resources) will be returned.

## 2.6 Client Server Architecture

When talking about a Client Server architecture, one refers to a system that is build by having a Client that is exposed towards the users and a Service which is only accessed by the Client. The Server normally does all the more power consuming work, such as storage handling or computation. The Client sends requests or data towards the Server which then processes the information and sends the result back

to the Client.

## 2.7 Diaspora

Diaspora [19] is an open source Social Network with properties similar to other well known SNS where you interact with friends, colleagues, family etc. registered on SNS. But unlike other SNS, Diaspora allows for more flexibility in some aspects, such as owning your own data, creating your own version of Diaspora and also not having a central server owned by a large company. The main reason for using Diaspora in this master thesis, is the flexibility and freedom in owning all the data and the service it self. This allows us to send and retrieve pretty much any data towards Diaspora.

# 3

# Architecture & Design

In Chapter 2 we presented the different policy strategies and the required technologies for developing a Dynamic Privacy Policy Framework (DPPF). With a combination of those we present a theoretical framework that can solve the problem described in Chapter 1. In this chapter we describe how we, in theory, solve the creation and evaluation of a DPP and present an architecture that implements this theoretical solution.

## 3.1 Dynamic Privacy Policy Framework

In this section we present our DPPF. The DPPF is divided into two parts, the creation of policies and the evaluation of policies. The first part shows a theoretical representation of the usage of DPP in the framework with the help of segmentation to later define how each segment contributes to the creation of a policy, and the second part shows in theory how the framework evaluates each policy.

### 3.1.1 Policy Creation

The first part of the framework is the creation of the policy. Assuming we have the following policy:

**Policy 3** *I don't want my colleagues to see my photos during weekdays between 08.00 and 17.00*

we can see for **whom** Policy is defined. We also see **what** type the policy is intended for and **when** the policy holds. Lastly we can also see if the policy **allows** or **denies** the audience group from viewing the content. These can be split into the four parts seen below.

| | |
|---|---|
| Who | my colleagues |
| What | photos |
| When | weekdays between 08.00 and 17.00 |
| Allow/Deny | don't want |

Every policy that is created will consist of four segments seen in Figure 3.1. Each segment has a property to define the structure of the policy and can be dynamic. Meaning each segment can have any type of value, e.g. segment group can be family, friends, etc. The framework will handle any combination of the segment since each

segment is evaluated as a single entity and only the combination of them can create a policy.

I don't want my colleagues to see my photos during weekdays between 08.00 and 17.00



**Figure 3.1:** Visual presentation of the segments in Policy 3

#### Group

In this thesis groups are defined as a set of audience predefined by the user. The audience must have a relationship with the user e.g. friends, family, etc. This is similar to how most SNS defines groups in their systems.

#### Permission

Permission is a key feature in both creating and evaluating a policy. This will let the user choose which groups will have the right to view or partake in the event that is being created by the user.
**NOTE:** *In this master thesis all policies will be negative policies. This means the policies will only define what groups are* denied *from viewing the content.*

#### Type

We define Type as the content of the event performed towards the SNS by the user. Types can be photo, message, tweet, etc. So when for example a user uploads a photo on the board, then *photo* is the type.

#### Context

A context is a data-model with two properties, the type of context and the data of the context. The context-type can be Time or Location and context-data is a dynamic representation of the context itself, e.g. a timestamp with some selected days or GPS location with a radius.

### 3.1.2 Policy Evaluation

The second part of the framework is the evaluation of the policies created. The evaluation will be executed when an event is created. An event is an action actively performed towards the SNS. Information necessary for the event to be processed will be sent, this will in turn help determine which group will have the rights to see the event.

Every policy saved for a user will be evaluated to find which policy applies to this particular event. The algorithm (see section 4.3) will then determine which policy is most suited and return the policy information with affected groups.

## 3.2 System Overview

In this section a short description of the system will be presented for a general understanding of the designed architecture. It consists of a client-side and server-side solution with different connected components. Each component in Figure 3.2 will be presented in more detail in the upcoming chapter.

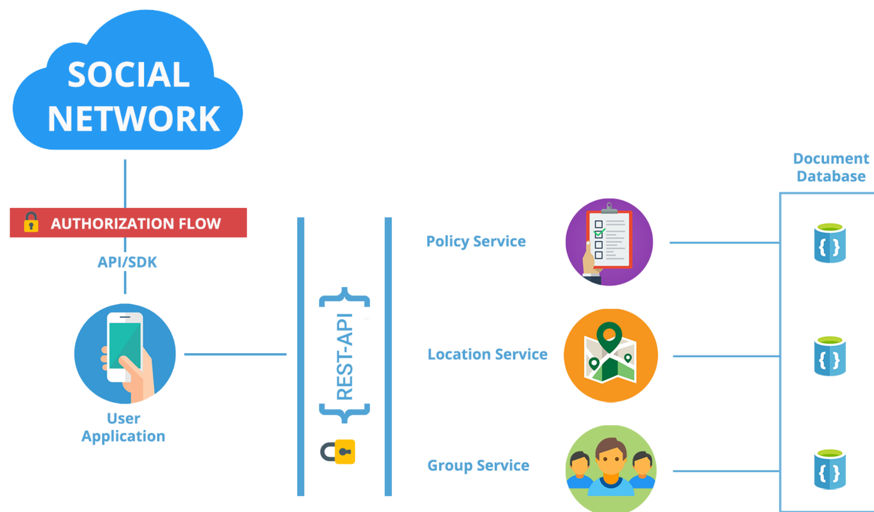



**Figure 3.2:** Overview of the System

The client-side of the system consist of a Mobile Application (User Application) with integration towards the different SNS. As previously mentioned in Chapter 1, each SNS provides an integration that allow the user to create content on the specific SNS. This integration can be implemented e.g. via an API or SDK. This is presented in Figure 3.2 as API/SDK and will be explained in depth in Section 4.2.4. Authorization of the user towards the SNS is implemented as required by the SNS and will be used to gain access to the API for a given user. This can differ between the SNS depending on what protocol is used and their implementation. This is presented as Authorization Flow in Figure 3.2 and will be explained in depth in Section 4.2.3. The component User Application (UA) is a Progressive Webb App (PWA). Every interaction with the user will be handle through the UA. Aside from creating and reading policies, the UA will also handle the evaluation of a policy when an event is created by the user. The UA also integrates with the server-side components through a secure RESTful-API, also known as a Service Integration.

The server-side of the system has a micro-service architecture. It consists of the following components Policy Service, Location Service and Group Service, with the option to easily increase with more services if needed. Each service uses Spring Boot Security [17] to provide a secured RESTful-API and is connected to a document

database for storing information.
Each service will provide the necessary information needed for the user application to make a successful evaluation and can be used, if needed, to operate on the data when big computations are needed.

The proposed architecture solution is designed with the building block concept in mind. When a new context, SNS or event-type is introduced, the system can easily increase the functionality with minimal change on the current code.

# 4

# Implementation

In the previous chapter we described the thought process behind the DPPF with details in how the DPP are handled and how they are evaluated within the DPPF. In this chapter we go through the implementation of the DPPF. We present in detail the different components and how they interact with each other to handle DPP. The algorithms used for evaluating DPP and handling conflicts between policies defined in the DPPF and SNS are also presented.

## 4.1 Server Side Implementation

This section describes the implementation of the server-side services presented in Figure 3.2. We present both the techniques and the logic behind each service.

### 4.1.1 Policy Service

The Policy Service is a Spring Boot [12] micro-service application written in Java 11 with RESTful API. The Policy Service main feature is to store the privacy policies defined by the user and sent from the User Application.
The incoming data towards the Policy Service is not altered before being stored in the database. However there is additional information added which is a key feature of the Policy Service. This is the creation of a readable string for the policy.

Requests made towards the Policy Service contain the following data in order to store or alter a policy:

- *userId* - This is used to specify which user the policy belongs to.
- *policyName* - The name of the policy which enables easier management for the policies.
- *groupId* - Group IDs for all the affected groups.
- *typeId* - All the different types for which the policy is defined.
- *permission* - This field defines whether the groups will be denied or allowed to view the content.
- *context* - The selected contexts for the policy.

The presented fields are defined in accordance with the specifications presented in Section 3.1.1. The JSON-structure below is a data-model presentation of the DPP

in our system with the addition of the owner of the policy and a given policy-name.

The requests looks as follow:

```
{
    "userId": "ExampleUser",
    "policyName": "MyFirstPolicy",
    "groupId": ["Family"],
    "typeId": ["Photo"],
    "permission" : "deny",
    "context": [
        {
            "gpsLocation": "Location1"
            "inside": False
        }
    ]
}
```

In order to create the readable string, fields in the incoming request are used. These are the *groupId*, *typeId* and *context*. These three fields will answer three key questions for a policy:

*Who* are blocked from viewing the content
*What* is blocked from being viewed
*When/Where* is the content blocked

This will in turn result to a readable string which reads:

**Policy 4** *I don't want my* Family *to see my* Photo *when I'm outside of* Location1

This readable string is then stored along with the other information. This makes the management of the policies easier for the user.

The following requests are available for Policy Service:

## Create Policy

**Policy Service Request 1** - POST [URL]/policy/addPolicy

```
{
   "userId": "ExampleUser",
   "policyName": "MySecondPolicy",
   "groupId": [
      "Family"
   ],
   "typeId": [
      "Photo"
   ],
   "permission" : "deny",
```

```
  "context": [
      {
          "gpsLocation": "Location1"
          "inside": False
      },
      {
          "from": "13:00",
        "to": "16:00",
        "days": [
           5,
           6,
           0,
        ]
      }
  ]
}
```

Since the context information is an array, it makes it easy for multiple contexts to be handled. The type is generic context type which allows for future contexts to easily be implemented.
We describe below the context for Time and Location.

### Time contextData

The contextData consists of three fields: *from*, *to* and *days*. *from* and *to* determines the time-interval between which the policy holds. The *days* field is an array with numerical representation of the different weekdays e.g. 0 = Monday, 6 = Sunday etc.

```
{
    "from": String,
    "to": String,
    "days" [Integer]
}
```

### Location contextData

The contextData only has two fields, *gpsLocation* and *inside*. The field *gpsLocation* is the name of the location corresponding with the name of the location stored in Location Service. The boolean field *inside* states whether the policy should hold when located inside or outside of *gpsLocation*.

```
{
    "gpsLocation": String,
    "inside": Boolean
}
```

## Fetch all policies for a User

**Policy Service Request 2** - GET [URL]/policy/getPolicies?

```
userId=ExampleUser
```

**Policy Service Request 2** - RESPONSE


```
{
   "userId": "ExampleUser",
   "policyName": "MySecondPolicy",
   "groupId": [
      "Family"
   ],
   "typeId": [
      "Photo"
   ],
   "permission" : "deny",
   "context": [
        {
            "gpsLocation": "Location1"
            "inside": False
        },
        {
            "from": "13:00",
            "to": "16:00",
            "days": [
                5,
                6,
                0,
          ]
        }
    ],
    "policyAsString": "I don't want my Family to see my Photo between
        13:00 and 16:00 during Monday, Saturday and Sunday and when I'm
        outside of Location1"
}
```


The response when fetching policies is similar to the information sent when storing policies. However only and perhaps the most important difference is the additional field *policyAsString* which is the generated readable string.

## Delete policies for a User

**Policy Service Request 3** - DELETE [URL]/deletePolicies?

```
userId=ExampleUser
&policyName=MyFirstPolicy
```

The Policy Service allows for removal of all policies for a user or one specific policy for a user.

### 4.1.2 Group Service

The Group Service is a Spring Boot micro-service application written in Java 11 with RESTful API for the exposed endpoints. The Group Service will handle the pre-defined groups for each respective SNS, these groups can later be changed to fetch the information from respective SNS. However each social network will have their own set of groups depending on the relationships. E.g family could consist of "Dad, Mom and Brother" in Facebook, while in Diaspora family consists of "Dad, Mom, Sister and Brother". Therefore each entry saved in the DB will have the group information and to which SNS this group belongs.

The Group Service allows for the following requests:

## Create/Update a group for a User

**Group Service Request 1** - POST [URL]/groups/addGroups

```
{
  "userId": "ExampleUser",
  "groups": [
      {
        "groupId": "7fghmgft68",
        "groupName": "Family",
        "groupMembers" : ["Dad","Mom","Brother"],
        "socialNetwork": "Facebook"
      },
      {
        "groupId": 5,
        "groupName": "Family",
        "groupMembers" : ["Dad","Mom","Sister","Brother"],
        "socialNetwork": "Diaspora"
      },
      {
        "groupId": "678gfd803g",
        "groupName": "Colleagues",
        "groupMembers" : ["Alice","Bob"],
        "socialNetwork": "Facebook"
      }
  ]
}
```

The variable groupId is the identification used for the groups on the different SNS. This means that future SNS can easily be implemented without to much effort. However the groupName is only used for the user to easily handle and maintain the

groups in the User Application.

### Fetch groups for a User

**Group Service Request 2** - GET [URL]/groups/getGroups?

```
userId=ExampleUser
```

**Group Service Request 2** - RESPONSE

```
{
  "userId": "ExampleUser",
  "groups": [
      {
         "groupId": "7fghmgft68",
         "groupName": "Family",
         "groupMembers" : ["Dad","Mom","Brother"],
         "socialNetwork": "Facebook"
      },
      {
         "groupId": 5,
         "groupName": "Family",
         "groupMembers" : ["Dad","Mom","Sister","Brother"],
         "socialNetwork": "Diaspora"
      },
      {
         "groupId": "678gfd803g",
         "groupName": "Colleagues",
         "groupMembers" : ["Alice","Bob"],
         "socialNetwork": "Facebook"
      }
  ]
}
```

The information is able to be fetched through the GET requests. Either all the groups for one user, or groups for a specific SNS through parameter `socialNetwork` e.g. &socialNetwork=Diaspora. This enables the User Application to be more flexible when handling an event.

### Delete group(s) for a User

**Group Service Request 3** - DELETE [URL]/groups/removeGroups?

```
userId=ExampleUser
```

Sending a DELETE request without `&socialNetwork=Diaspora` parameter will re-

sult in removing all the groups for a user regardless of SNS. Removing groups for a specific SNS requires the SNS to be defined in as a path parameter.

### 4.1.3 Location Service

Similar to the Group and Policy Service, the technical setup of the Location Service is a micro-service Spring Boot application written in JAVA 11 with RESTful api. The Location Service is responsible for handling and storing the defined locations for a user. Unlike the Group Service, the stored information is not SNS dependent.

The Location Service allows for the following requests:

## Add/Update location

**Location Service Request 1** - POST [URL]/location/saveLocation

```
{
    "userId": "ExampleUser",
    "locationData": [
        {
            "locationName": "Location1",
            "gpsLocation": {
                "lat": 57.733893,
                "lon": 12.031323
            },
            "radius": 1000
        }
    ]
}
```

The provided body when saving a location includes the user for who the location belongs along with locationData consisting of locationName, the gps location and the radius for indicating the size of the location.

## Fetch all locations for a User

**Location Service Request 2** - GET [URL]/location/getLocations?

```
    userId=ExampleUser
```

The get request returns all the stored locations for a user.

**Location Service Request 2** - RESPONSE

```
{
    "userId": "ExampleUser",
    "locationData": [
```

```
        {
            "locationName": "Location1",
            "gpsLocation": {
                "lat": 57.733893,
                "lon": 12.031323
            },
            "radius": 1000
        },
        {
            "locationName": "Location2",
            "gpsLocation": {
                "lat": 58.733893,
                "lon": 11.031323
            },
            "radius": 2000
        }
    ]
}
```


Important to note is that no alteration is done on the incoming data before being stored in the database. Therefore the GET response looks similar to the POST request in terms of data.

### Delete location(s) for a User

**Location Service Request 3** - DELETE [URL]/location/removeLocations?

```
    userId=ExampleUser
```

Similar to the DELETE request in the Group Service, sending a DELETE request without `locationName` parameter will result in removing all the locations for a user. Removing a specific location requires the `locationName` to be sent as a path parameter.

## 4.2 Client Side Implementation

This section describes the implementation of the client-side application. It will cover the implementation towards the server-side services, the Authorization Flow and the integration towards the different SNSs.

### 4.2.1 User Application

As previously explained, the User Application will be used to create policies and content on the SNS and to evaluate the policies when an event is created.

The application itself is a Progressive Web App (PWA). By using a PWA we support both Android and iOS with the same code-base. The development is done in JavaScript by using Vue.js [13] as a framework with Vuetify [14] as a UI-Library.

The architecture is set to a Component Design Architecture, this allows us easily to a reuse same UI-components on different view-models. Due to the support of the server-side services the application is stateless and does not save any client data that can be reused during the next session.

### 4.2.2 Server Side Integration

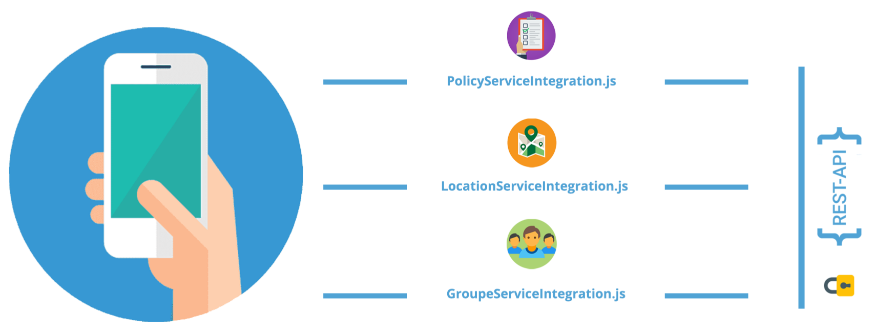



**Figure 4.1:** An overview of the integration towards the different services.

The User Application is connected to the different micro-services that exists. To make the implementation as dynamic and secure as possible each micro service from the server side is represented as a integration service in the User Application. This makes future changes easy to implement. All request are done with a given token provided from the Spring Security [17] implementation.

Each integration is responsible to transform and request data from the existing micro services. Meaning that if data needs to be transformed into a specific model, required by a view component or module, the integration service will handle the task. However, the integration services will not handle component or module specific logic.

### 4.2.3 Authorization Flow

#### 4.2.3.1 Diaspora

For Diaspora, we use OpenID Connect for dynamic client registration, discovery service and for the authorization flow. Following the concept of OAuth 2.0 described in section 2.5, the following is done for Diaspora.

**Application Registration**
A developer needs first to register the application on Diaspora. This is done over the OpenID-API by making a HTTP-POST request with the following body:

**Application Registration Request 1** - POST [URL]/api/openid_connect/clients

```
{
  "client_name": "ExampleApp",
  "redirect_uris": [
    "http://example.com/"
  ]
}
```

This will return a response with the data needed for the Authorization and Token Request, client_id and client_secret.

**Application Registration Request 1** - RESPONSE

```
{
    ...
    "client_id":"b619f6868684996322b059a101b88bf7",
    "client_secret":"f17658e85ff0bfc7aa615d77528ed919
    317ae53902fa8a21c9b8b233753aaae5",
    "client_name":"ExampleApp",
    ...
}
```

**Getting Permissions**
In the next phase the user needs to grant our application for the requested permissions on Diaspora. This is done by forwarding the user to Diaspora via a generated query-string with following parameters:

**Get Permission Request 1** - GET [URL]/api/openid_connect/authorizations/new?

```
    response_type=code
    &scope=openid+contacts:read+contacts:modify+profile:read...
    &redirect_uri=http%3A%2F%2Fexample.com%2F
    &client_id=b619f6868684996322b059a101b88bf7
```

| | |
|---|---|
| **response_type** | defines the type of flow for the specific implementation |
| **scope** | a list of permissions that our application request, e.g. read contact list |
| **redirect_uri** | the URL we forwarded to when the user denies or approves the request |
| **client_id** | the Id of the application |

The user now needs to approve or deny permissions as seen in Figure 4.2 via the dialog from Diaspora.

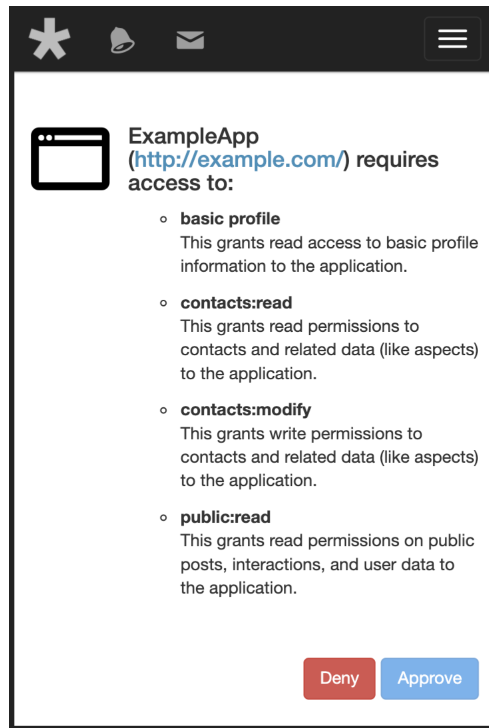



**Figure 4.2:** Diaspora Permission Dialog

If approved the user will be redirected back to the application with a Grant Token provided in a query-parameter (?code), this will be used for requesting the Access Token.

**Get Permission Request 1** - RESPONSE

```
"http://example.com/?code=0a00caf8afef9807e04b0460c5a
    af520dc343262f87d072755b58668fb607ef4"
```

### Getting Access Token

Now with permission granted we make a request to Diaspora and request access token. This is done with a POST request with the following query-parameters:

**Get Token Request 1** - POST [URL]/api/openid_connect/access_tokens?

```
grant_type=authorization_code
&code=0a00caf8afef9807e04b0460c5a
    af520dc343262f87d072755b58668fb607ef4
&redirect_uri=http%3A%2F%2Fexample.com%2F
&client_id=b619f6868684996322b059a101b88bf7
&client_secret=f17658e85ff0bfc7aa615d77528ed919
    317ae53902fa8a21c9b8b233753aaae5
```

The response is a JSON object with the following content.

**Get Token Request 1** - RESPONSE

```
{
    "access_token":"82db50dcd19234c488d0e0b86c7d727e7
        fce405e80b3f271de48e256eac398aa",
    "refresh_token":"a80bab733c2f226d439c129c810c0f6d
        0bb7ac258aefb84ce27d4a368ad87980",
    "token_type":"bearer",
    "expires_in":"86399",
    "id_token":"eyJ0eXAiOiJKV1QiLCJhbGciOi...jWuBp8NXMCNXxUWD2B435gRPF"
}
```

The Access Token is now valid for 24 hours and can be refreshed with the Refresh Token for another 24 hours. The following options can now be used to transmit the access token when requesting a resource from Diaspora:

- as a query-parameter: [URL]/api/v0/example?access_token=[access token]
- as an application/x-www-form-urlencoded parameter: access_token=[access token]
- inside the request header: Authorization: Bearer [access token]

#### 4.2.3.2 Facebook

For Facebook, we use the "Login Dialog" provided via the Facebook JavaScript SDK in the authorization flow. Important to know is that Facebook also supports manually implementation of the login flow [6].

**Application Registration**
A developer needs to register the application as an app via the "Facebook for Developer" site. This process consists of a graphical wizard with a set of configuration desired for the application. After registration the app will be provide with an app-id that is used for communication in the JavaScript SDK.

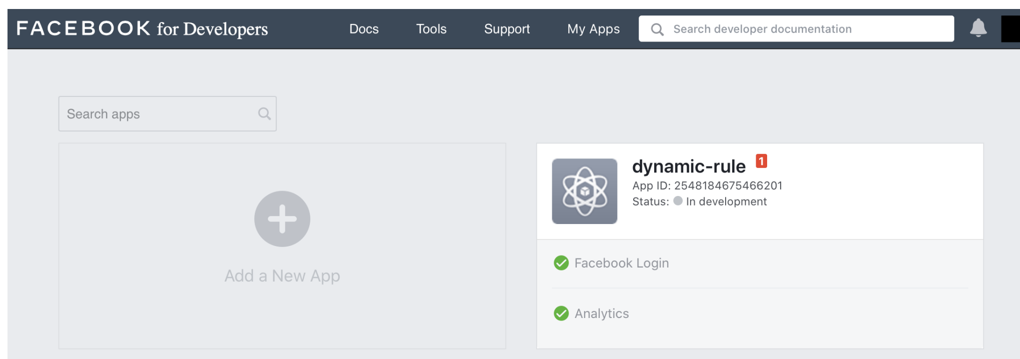



**Figure 4.3:** Facebook for Developer

**Getting Permissions and Access Token**

Using the JavaScript SDK this process requires only a few line of code and the user approval. The JavaScript SDK will open the login-dialog from Facebook and ask for the permission configured by the request. The implementation and dialog are shown below:

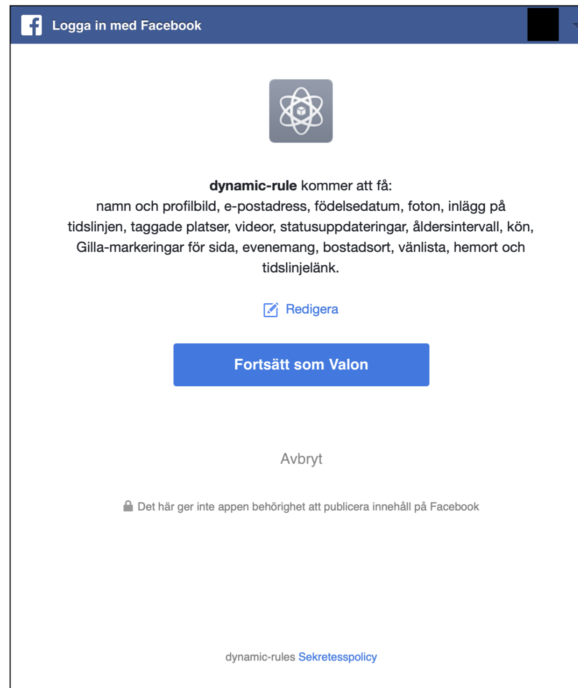



**Figure 4.4:** Facebook Login Dialog

**Permission and Token Request 1** - JavaScript SDK FB.login()

```
FB.login(function(response) {
  // handle the response
}, {scope: 'public_profile,email,user_friends...'});
```

The login method via the SDK will handle the dialog seen in Figure 4.4 and the response from Facebook. Permissions to the resource are defined via the **scope** object in the request. The developer of the application can specify what permissions are needed, e.g. the application should be able to see profile-picture and name of the user. A list of available permissions are presented in the official documentation [7].

**Permission and Token Request 1** - RESPONSE

```
{
    status: 'connected',
    authResponse: {
        accessToken: '{access-token}',
        expiresIn:'{unix-timestamp}',
        reauthorize_required_in:'{seconds-until-token-expires}',
        signedRequest:'{signed-parameter}',
```

```
        userID:'{user-id}'
    }
}
```

If approved the response will contain the Access Token as shown in the code example and is ready to be used with the Facebook Graph API for requesting resources.

### 4.2.4 Social Network Integration

To interact with a SNS, the official API or SDK of the SNS is used. Different API and SDK are provided with different design. Each API or SDK are mapped to a integration service in the user application.

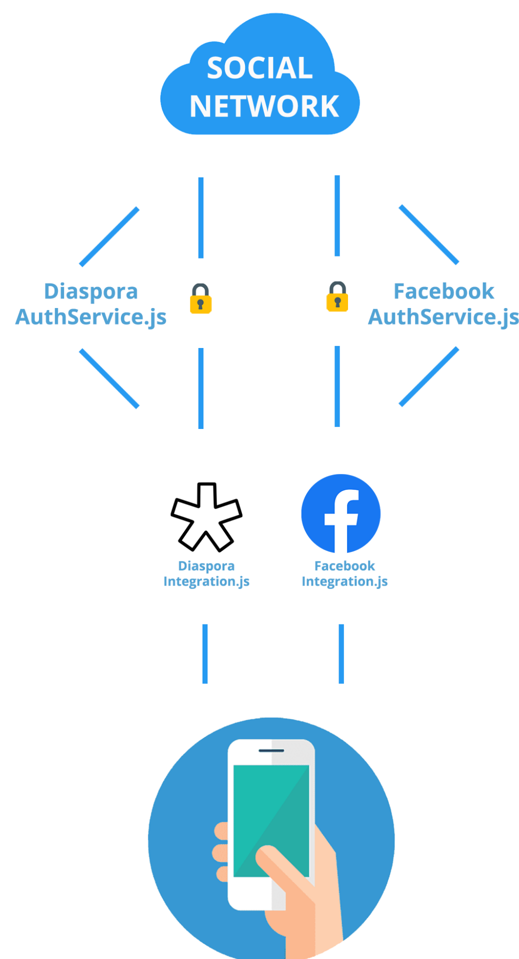



**Figure 4.5:** Overview of the Social Network Integration

If no tokens exist in the application, the user will need to use the designed Auth-Service for the specific SNS shown in Figure 4.5. Again, this approach makes the system dynamic in terms of a new or modified integration.

#### 4.2.4.1 Diaspora API

For Diaspora, Diaspora API v1 is used. The API will be supported in the future release of Diaspora v0.8.0.0. Below is an example of how to publish a post on Diaspora using the API.

**Diaspora Request 1** - POST /api/v1/posts
Required permission: private:modify

```
{
  "body": "Answer is always 42!",
  "public": false,
  "aspects": [
    6,
    7
  ]
}
```

Documentation for all supported parameters can be found on the official documentation [18]. The Diaspora Integration in our application supports a one-to-one data-mapping at the current state.

#### 4.2.4.2 Facebook API and SDK

For Facebook, Facebook Graph API v7.0 is used. The API is a RESTful API and is used for creating and uploading content to your profile, group or page. Below is an example on how to publish a post on Facebook using the API or SDK

**Facebook API Request 1** - POST https://graph.facebook.com/me/feed?
Required permission: pages_read_engagement, pages_manage_posts

```
message=Answer%20is%20always%2042!
&access_token={access-token}"
```

**Facebook SDK Request 1** - JavaScript SDK (after initialize)
Required permission: pages_read_engagement, pages_manage_posts

```
FB.api(
  '/me/feed',
  'POST',
  {"message":"Answer is always 42!"},
  function(response) {
      // handle response
  }
);
```

Documentation of all parameters can be found on the official documentation.

## 4.3 Algorithm

This section explains the algorithms used for creating an event, for evaluating the stored policies and for handling conflicts towards policies in the social network.

### 4.3.1 Evaluation Algorithm

The evaluation algorithm presented below is used when a user creates an event, both to evaluate and to find a policy that affect the event. All defined policies include groups which are needed to set the correct audience group on the outgoing events.

The evaluation algorithm is presented in pseudo code below:

```
{
    ARGUMENT eventType, userPolicyList

    list <- []

    FILTER userPolicyList by eventType

    IF userPolicyList is empty
    THEN
        RETURN list

    FOR each policy in userPolicyList
        "call evaluation method for each context in the policy"
        IF validation pass
        THEN
            ADD policy to list

    RETURN list

}
```

After an event has been triggered, the evaluation algorithm is run. The algorithm takes the eventType $E$ (e.g. uploading a picture) as input along side all the defined policies for the user. It will then iterate through all the policies to filter out the once that hasn't $E$ defined. After that the algorithm evaluates each context present in the filtered policies, with the help of the evaluation methods presented below, to even further filter out the policies. This will in turn leave us with policies that will be enforced for the triggered event.

#### 4.3.1.1 Evaluation Methods

The implemented context has the following evaluation methods needed for evaluating the context data present in each policy. Since each context have their own evaluation method, it is easy to add new context without modifying the evaluation algorithm.

***Time Context***

```
{
   FUNCTION timeContextEvaluation ARGUMENT contextData

   now <- "current timestamp"
   from <- contextData.from
   to <- contextData.to

   IF "desired days does not include current day"
   THEN
      RETURN false
   ELSE
      RETURN "from < now < to"
}
```

The evaluation method for Time Context takes the `contextData` as argument. The method first checks if the current day is included in the list of selected days for the policy. If it is included then the evaluation checks if the current timestamp is between the time defined by the policy.

***Location Context***

```
{
   FUNCTION locationContextEvaluation ARGUMENT contextData

   currentLocation <- "current GPS location"
   policyLocation <- "fetch location by contextData.location from
      Location Service"

   inside <- FUNCTION insideCircle(
      currentLocation, policyLocation.center, policyLocation.radius)

   RETURN (inside AND policyLocation.within)
}
```

The evaluation method for Location Context takes the `contextData` as argument. The method checks if our current location resides within a defined radius of the location in the policy. The method will then return true or false, depending on the location definition for the policy.

### 4.3.2 Policy Priority Algorithm

After the evaluation algorithm return all affecting policies, the application needs to ensure that no policy on the SNS affect the event. Meaning that only the polices from the application will be applied on the event.

```
{
   ARGUMENT evaluatedPolicies,socialNetworkPolicies

   FOR each policy in socialNetworkPolicies
      "Check for conflicts between policy and evaluatedPolicies
         affecting the same group and event"
      IF conflict exists
      THEN
         "Notify user about conflict"
}
```

Given that the SNS provides context-data similar to the definition of our context-data, e.g. Facebook provides from, to and days in their policy data with regards to Time Context, we can execute the previous explained context evaluation methods on the Facebook context-data. If the evaluation methods output the same result as for the evaluated policies, then no conflict exists.

# 5
# Results

In this chapter, we present the result of validating the different goals of this master thesis. The results include the centralization of the policy management for multiple SNS, the usage of multiple contexts on a privacy policy and the privacy concern of contextual information for the users. The results of the implemented application are presented in Chapter 4.

## 5.1 Predefined scenario

In order to test the implementation, we defined a scenario mimicking real life. We used user Alice as focus point for the test with all relationships going outwards from Alice. The locations and groups were all defined from Alice perspective.
In this scenario Alice has 3 groups for which Alice has some relationship with the users. These are, family with Bob through Facebook, family with Bob and Charlie through Diaspora, and colleagues with David and Evan through Diaspora. The stored locations for Alice are home and work.
The data defining this scenario can be seen below.

**Locations**:

```
{
    "userId": "Home",
    "gpsLocation": {
        "lat": 57.692163,
        "lon": 11.949058
    },
    "radius": 1000
}
```

```
{
    "userId": "Work",
    "gpsLocation": {
        "lat": 57.708082,
        "lon": 11.961515
    },
    "radius": 1000
}
```

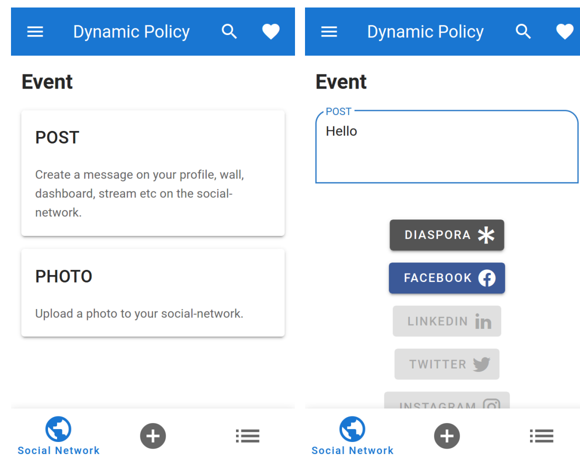
**Groups/Relationships**:


```
{
    "userId" : "Alice",
    "groups" [
        {
            "groupId": "7fghmgft68",
            "groupName": "Facebook Family",
            "groupMembers": ["Bob"],
            "socialNetwork": "Facebook"
        },{
            "groupId": "5",
            "groupName": "Diaspora Family",
            "groupMembers": ["Bob","Charlie"],
            "socialNetwork": "Diaspora"
        },{
            "groupId": "6",
            "groupName": "Diaspora Colleagues",
            "groupMembers": ["David","Evan"],
            "socialNetwork": "Diaspora"
        }
    ]
}
```


## 5.2 Multiple Social Networks

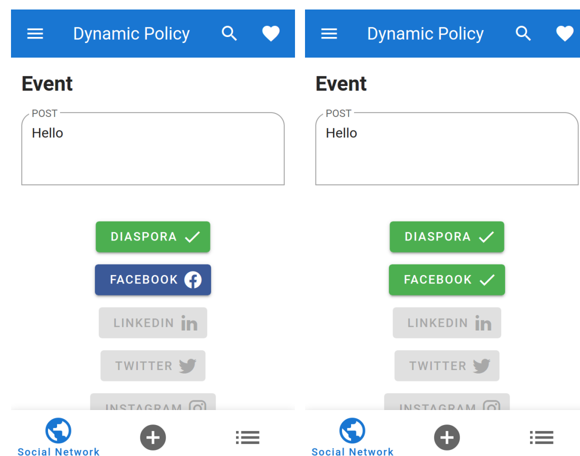
In order to get the result for the usage of multiple SNS simultaneously, an implementation towards different SNS was needed. We implemented integration with Facebook and Diaspora. This made it possible for us to create outputs for each SNS and send them respectively.

In Figure 5.1b we can see the integration towards the different SNS in the application. A single defined policy can be used on all supported SNS without the need for further configuration. Each integration will use this policy to create a request towards the SNS that follows their specification of creating content with the defined policy applied. This can be seen in Figure 5.2a-5.2b with the output data in Figure 5.2c-5.2d.

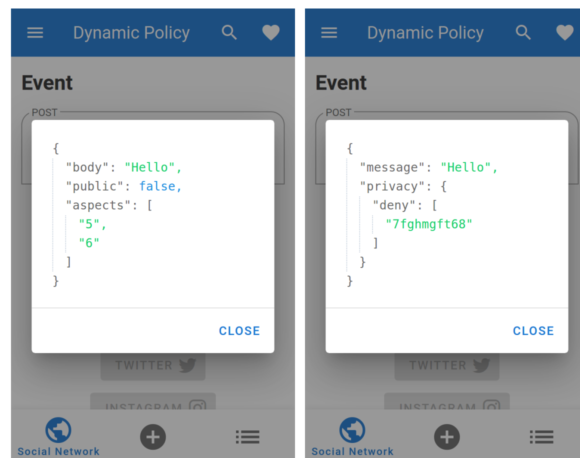



(a) Choose an event. (b) Choose SNS to send to.

**Figure 5.1:** Triggering an event




(a) Post to Diaspora (b) Post to Facebook




(c) Output for Diaspora (d) Output for Facebook

**Figure 5.2:** Posting to multiple SNS

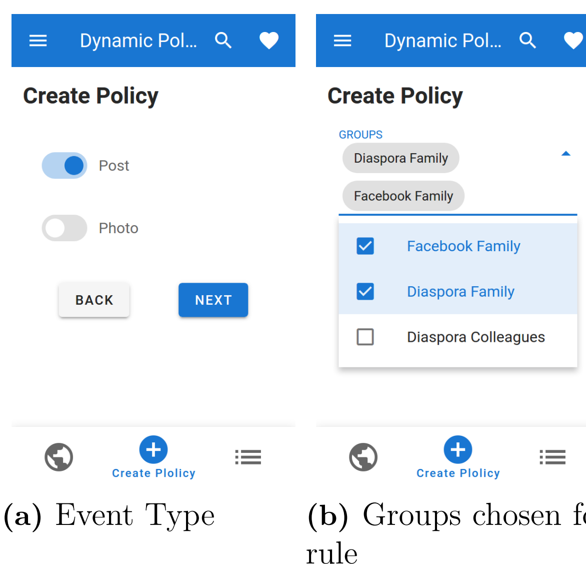
(a) Diaspora Stream (b) Facebook Response

**Figure 5.3:** The post represented in each SNS.

Shown in Figure 5.3a is the Diaspora stream with the posted message. However, in Figure 5.3b the post was not successfully shown in Facebook due to the limitations of the API which we will discuss in Chapter 6.

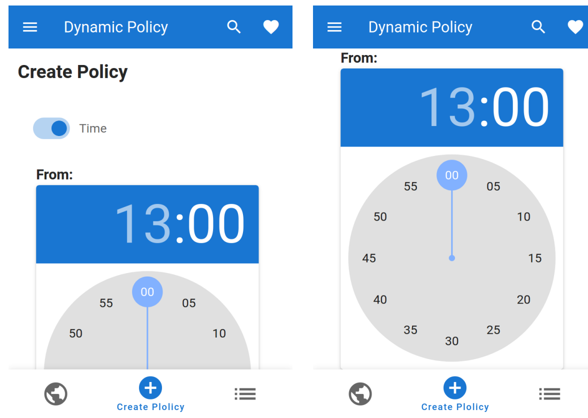
## 5.3 Multiple Context

The application allows users to use multiple contexts in one single privacy policy. This means that the application will take all contexts into account when posting.

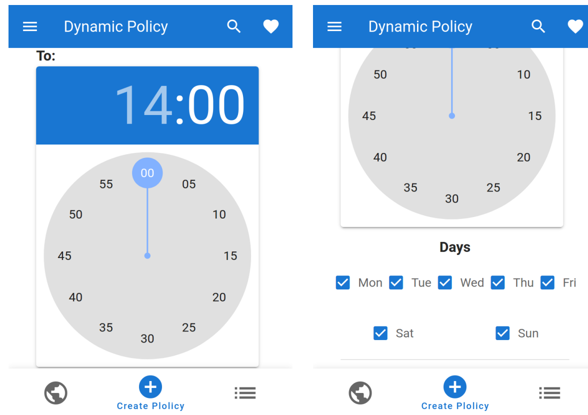



(a) Event Type (b) Groups chosen for rule

**Figure 5.4:** Choose event type and groups for which the policy should apply

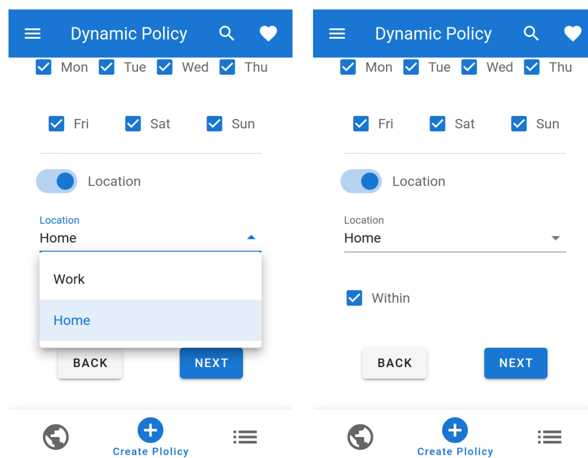



(a) Time context chosen (b) Chosen From time



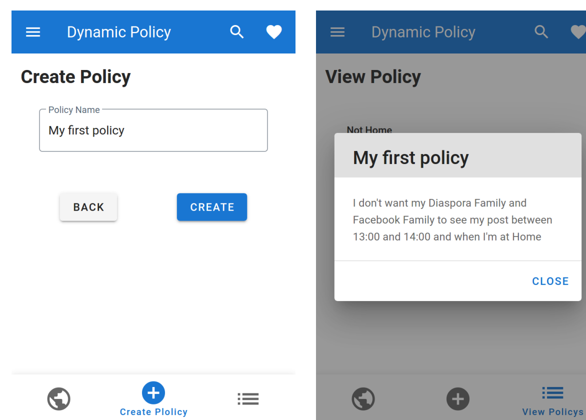

(c) Chosen To time (d) Chosen Days

**Figure 5.5:** Choose variables for Time context




(a) Chosen Location (b) Within boolean enabled

**Figure 5.6:** Choose variables for Location context




**(a)** Name of the policy **(b)** Readable String

**Figure 5.7:** Naming the policy and viewing the readable string

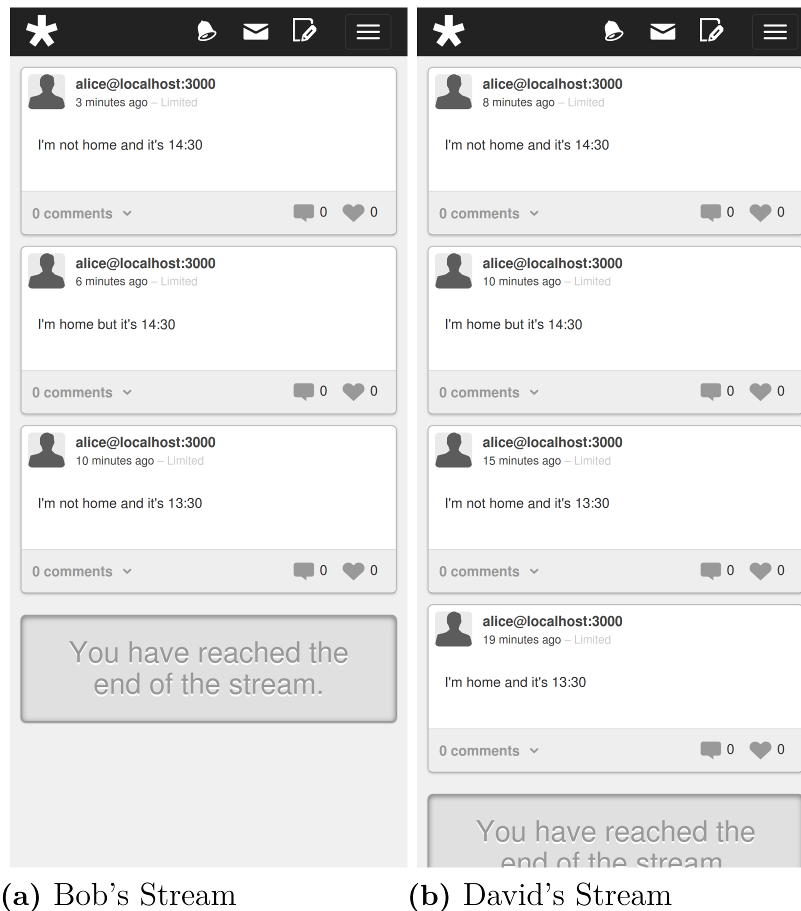
Figures 5.4 - 5.7 show how a rule can be created with more than one context included. The readable string outputs the following string:

*I don't want my Diaspora Family and Facebook Family to see my post between 13:00 and 14:00 and when I'm at Home*

Given the rule created in 5.4 we can show four post scenarios from Alice shown on Bob's (Family) and David's streams:

- *Scenario 1* - Alice is at home and time is 13.30
- *Scenario 2* - Alice is not at home and time is 13.30
- *Scenario 3* - Alice is at home and time is 14.30
- *Scenario 4* - Alice is not at home and time is 14.30

These scenarios will show the enforcement of the rule in terms of audience groups.




**(a)** Bob's Stream **(b)** David's Stream

**Figure 5.8:** Bob's and David's Stream with Alice posts after all scenarios

What we can clearly see is that all posts made by Alice are not visible in Bob's stream. These are however visible in David's stream. The policy states that no Diaspora Family members should be able to see posts made by Alice when the policy holds, which is clearly shown in Figure 5.8. It also shows all the posts where the policy does not hold.

## 5.4 Securing Contextual Information

The requests towards any of the services do not include any contextual information from the user. This information is instead only used locally on the client in the algorithms explained in Section 4.3. The requests for each service can be seen in Section 4.1 Server Side Implementation.
Securing the contextual information is made possible by having the evaluation algorithm being run on the client side and having a stateless implementation.

# 6

# Discussion

The result show how the developed implementation validates the master thesis three main research questions: *How do we centralize dynamic privacy policy management of multiple SNS?*, *How can we make use of multiple contexts?* and *How do we secure the users contextual information?*.

### How do we centralize dynamic privacy policy management of multiple SNS?

The results shown in section 5.2 indicate that it is fully possible to centralize the management for privacy policies for multiple SNS. What the outputs in Figure 5.2c-5.2d show is that the possibility of sending an event to multiple SNS simultaneously is very much possible, along with the possibility to add more SNS with little effort.

This is possible to do as long as the SNS provides an API or SDK that allows external integration. In Figure 5.3a we can clearly see that the post has reached Diaspora and posted on the User Stream, this is possible since Diaspora provides such an API. Looking at the response from Facebook in 5.3b, we can see that we are missing permissions for publishing. Facebook has since the 24th of April 2018 [11] removed permissions to publish on User Feed.

However, there might be a workaround to this limitation. Using a browser plug-in/extension, one could, in theory, modify the request sent from the browser. This would result in a plugin that the DPPF would communicate with before executing the request towards the SNS.

---

```
Sample of the payload:
{
    ...
    "privacy" : //policy-settings defined by Facebook
   "scope":
     {
        "label":"Chosen friends",
        "can_viewer_edit":true,
        "description":"Arbnor Biljali",
        "extended_description":"Valon has chosen a custom audience for
           this post"
     }
   ...
```

```
}
```

A request could in theory be modified via the DPPF and result in a modified payload where audience and permission would be provided by the DPPF. In this thesis we focused on an centralized implementation and did therefore not explore the possibilities of using plugin/extension for the different browsers. Also, this approach is extremely time-consuming.

**How can we make use of multiple contexts?**

With regards to the paper by Pardo et al. we extend their work by adding two new additional context to be used.
While they use server side time we make use of client side time context. This enables for better and more accurate definitions of the privacy policies. We also added the GPS location as context to be used. By adding these two new contexts we allow for more flexible constructions, making it easier for users to have more expressive policies.
Our result show that we can use multiple context in a privacy policy. This does however not exclude the fact that one can use one context in their privacy policy.

**How do we secure the users contextual information?**

When it comes to the users contextual information, as shown in the result, no information is revealed outside of the client. This was done by choosing the stateless implementation, due to the fact that no information about of the session leaves the client. This however comes with some constraints, in an event of client- or connection error, the session will have to restart since the state is not shared or stored among the different clients.
Furthermore, all requests made towards the backend-services, results in receiving all the data stored for the specific user. By doing this we move the risk of knowing patterns about the users whereabouts and most frequent communications.

## 6.1 Conclusion

The thesis was validated with the application described in Chapter 4. The result agree with the implemented architecture with regards to multiple context and centralized management, and as long as there exists a SNS with API/SDK for external integration, an implementation is possible.

The policy priority algorithm could not be validated towards a SNS because of two reasons. One is that Diaspora, which we have full integration with, has no policy management support. The second reason is that Facebook has a very strict API which doesn't allow for the application to retrieve policies defined in Facebook.

## 6.2 Future Work

To further make use of dynamic privacy policies one could consider to take historical events into account. By this we mean that when altering or adding a new dynamic privacy policy then this should somehow propagate to the previously triggered event and changing their audience group respectively.

Another interesting extension of this master thesis would be to examine the possibility to use a mixture of both allow and block nature when defining a privacy policy.

Lastly we would recommend that one should take a look at an alternative architecture where all data management and operations are done on the client side with a stateful implementation.

# Bibliography


[1] Yabing Liu, Balachander Krishnamurthy, Balachander Krishnamurthy and Alan Mislove. *Analyzing Facebook Privacy Settings: User Expectations vs. Reality.*
https://www.ccs.neu.edu/home/amislove/publications/Privacy-IMC.pdf

[2] Matthew Rosenberg et al. *https://www.nytimes.com/2018/03/17/us/politics/cambridge-analytica-trump-campaign.html* , 2018-03-18

[3] Raul Pardo, Christian Colombo, Gordon J. Pace and Gerardo Schneider. *"An Automata-based Approach to Evolving Privacy Policies for Social Networks".*
http://www.cse.chalmers.se/∼gersch/rv2016.pdf

[4] Jürgen K. Müller *The Building Block Method.*
https://www.gaudisite.nl/proefschrift_jkm_bbm.pdf

[5] Facebook Login, https://developers.facebook.com/docs/facebook-login/, accessed: 2020-06-04

[6] Facebook Manually Login Flow, https://developers.facebook.com/docs/facebook-login/manually-build-a-login-flow 2020-06-10

[7] Facebook Permission Reference, https://developers.facebook.com/docs/permissions/reference 2020-06-10

[8] OpenID Connect, https://openid.net/connect/, accessed: 2020-06-04

[9] Sign in with Twitter, https://developer.twitter.com/en/docs/basics/authentication/guides/log-in-with-twitter, accessed: 2020-06-04

[10] author = Blank, Grant and Bolsover, Gillian and Dubois, Elizabeth, year = 2014, month = 04, pages = , title = A New Privacy Paradox: Young People and Privacy on Social Network Sites, journal = SSRN Electronic Journal, doi = 10.2139/ssrn.2479938

[11] Facebook User Feed API Documentation *https://developers.facebook.com/docs/graph-api/reference/v8.0/user/feed*

[12] Spring Boot *https://spring.io/projects/spring-boot*, accessed: 2020-07-20

[13] Vue *https://vuejs.org/*, accessed: 2019-08-22

[14] Vuetify *https://vuetifyjs.com/en/*, accessed: 2019-08-22

[15] Micro-service Architecture *https://www.redhat.com/en/topics/microservices/what-are-microservices*, accessed: 2020-07-22

[16] Restful Api *https://restfulapi.net/rest-architectural-constraints/*, accessed: 2020-07-27

[17] Spring Boot Security *https://spring.io/projects/spring-security*, accessed: 2020-07-20

[18] Diaspora API v1 *https://diaspora.github.io/api-documentation/*, accessed: 2020-07-20

[19] Diaspora Social Network *https://diasporafoundation.org/*, accessed: 2020-10-19