Comenius University, Bratislava
Faculty of Mathematics, Physics and Informatics

Integrated digital user profiles

Master's Thesis

2016 Bc. Mária Šormanová

Comenius University, Bratislava
Faculty of Mathematics, Physics and Informatics

Integrated digital user profiles

Master's Thesis

Study programme:	Computer Science
Study field:	2508 Computer Science
Study department:	Department of Computer Science
Supervisor:	RNDr. Martin Homola, PhD.
Consultant:	doc. RNDr. Zuzana Kubincová, PhD.

2016 **Bc. Mária Šormanová**

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname:	Bc. Mária Šormanová
Study programme:	Computer Science (Single degree study, master II. deg., full time form)
Field of Study:	Computer Science, Informatics
Type of Thesis:	Diploma Thesis
Language of Thesis:	English
Secondary language:	Slovak

Title: Integrated digital user profiles

Aim: Propose a method to integrate profile data related to a selected user created and published independently using several standalone web services. Implement the proposed method into a working system.

Literature:

Ko, Moo Nam, Gorrell P. Cheek, Mohamed Shehab, and Ravi Sandhu. "Social-networks connect services." Computer 43, no. 8 (2010): 37-43.

Bodle, Robert. "Regimes of sharing: Open APIs, interoperability, and Facebook." Information, Communication & Society 14, no. 3 (2011): 320-337.

Mika, Peter, and Tim Potter. "Metadata Statistics for a Large Web Corpus." LDOW 937 (2012).

Guha, Ramanathan. "Introducing schema. org: Search engines come together for a richer web." Google Official Blog (2011).

Annotation: Web users are currently using a number of different web services such as Facebook, Twitter, Blogger, Quora, Stack Overflow, Github, etc. With each service they create a part of their digital identity. However, these parts are separated and disconnected from each other. The task of this diploma thesis is to come up with a unifying and, as much as possible, standard way how to integrate these standalone identities into a coherent user profile. The goals will be achieved by exploiting the existing rich meta data standards such as RDFa, Schema.org, OpenGraph, Twitter cards, etc., and extending them for sake of the given task. The aim is to come with such a method of profile data integration that requires zero or minimal additional effort from services that are already employing these meta data standards to annotate the output of the users.

Supervisor:	RNDr. Martin Homola, PhD.
Consultant:	doc. RNDr. Zuzana Kubincová, PhD.
Department:	FMFI.KAI - Department of Applied Informatics
Head of department:	prof. Ing. Igor Farkaš, Dr.
Assigned:	16.12.2014

Approved: 17.12.2014 prof. RNDr. Branislav Rovan, PhD.
Guarantor of Study Programme

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

....

Student Supervisor

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Bc. Mária Šormanová
Študijný program: informatika (Jednoodborové štúdium, magisterský II. st., denná forma)
Študijný odbor: informatika
Typ záverečnej práce: diplomová
Jazyk záverečnej práce: anglický
Sekundárny jazyk: slovenský

Názov: Integrated digital user profiles
Integrovaný digitálny používateľský profil

Cieľ: Navrhnúť metódu pre integráciu profilových dát, vzťahujúcich sa na určitého používateľa, ktoré publikuje nezávisle pomocou viacerých samostatných webových služieb. Navrhnuté riešenie implementovať do funkčného systému.

Literatúra:
Ko, Moo Nam, Gorrell P. Cheek, Mohamed Shehab, and Ravi Sandhu. "Social-networks connect services." Computer 43, no. 8 (2010): 37-43.

Bodle, Robert. "Regimes of sharing: Open APIs, interoperability, and Facebook." Information, Communication & Society 14, no. 3 (2011): 320-337.

Mika, Peter, and Tim Potter. "Metadata Statistics for a Large Web Corpus." LDOW 937 (2012).

Guha, Ramanathan. "Introducing schema. org: Search engines come together for a richer web." Google Official Blog (2011).

Anotácia: Používatelia webu využívajú množstvo webových služieb ako Facebook, Twitter, Blogger, Quora, Stack Overflow, Github, atď. Pomocou týchto služieb si vytvárajú časť svojej digitálnej identity. Tieto časti sú však oddelené a nezávyslé jedna od druhej. Cieľom diplomovej práce je preto prísť so zjednocujúcim spôsobom, ako spojiť tieto časti a vytvoriť koherentný používateľský profil. Pre dosiahnutie tohto cieľa by sme chceli čo najviac využiť existujúce metódy na anotáciu meta dát ako RDFa, Schema.org, OpenGraph, Twitter cards, atď., a rozšíriť ich pre naše potreby. Ambícia je prísť s metódou integrácie používateľských profilov, ktorá vyžaduje minimálne alebo žiadne úsilie zo strany služieb, ktoré už podporujú niektorý spôsob anotácie obsahu, produkovaného svojimi používateľmi.

Vedúci: RNDr. Martin Homola, PhD.
Konzultant: doc. RNDr. Zuzana Kubincová, PhD.
Katedra: FMFI.KAI - Katedra aplikovanej informatiky
Vedúci katedry: prof. Ing. Igor Farkaš, Dr.

Dátum zadania: 16.12.2014

Dátum schválenia: 17.12.2014

prof. RNDr. Branislav Rovan, PhD.
garant študijného programu

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

....

študent vedúci práce

Acknowledgements

I would like to thank my supervisor RNDr. Martin Homola, PhD. and consultant doc. RNDr. Zuzana Kubincová, PhD. for all their help, advice and support during writing of this thesis.

I hereby declare that I wrote the thesis *Integrated digital user profiles* by myself, only with the help of the referenced resources, under the careful supervision of my thesis supervisor.

Bc. Mária Šormanová

Abstrakt

Používatelia webu používajú veľké množstvo rozličných služieb. Vo väčšine z nich o sebe zdieľajú informácie a vytvárajú si svoj používateľský profil – svoju *digitálnu identitu*. Jeden človek v dnešnej dobe vlastní desiatky takýchto používateľských účtov, čo je častokrát náročné na administráciu. Jednotlivé služby spolu kominikujú len minimálne a používateľ musí spravovať každý svoj účet samostatne. Existujúce systémy riešia tieto problémy len čiastočne. Cieľom tejto práce bolo navrhnúť a implementovať nový systém, ktorý používateľovi uľahčuje manažment a správu svojej digitálnej identity na jednom mieste. V prvej časti práce staviame na webovom autentifikačnom protokole OpenID Connect, do ktorého sme navrhli rozšírenie. To umožňuje širokú a efektívnu komunikáciu medzi zapojenými webovými službami. Jednou z hlavných výhod je možnosť používateľa definovať automatickú aktualizáciu profilových údajov na službách ktoré používa. V druhej časti práce sme navrhli systém na správu používateľských identít. Používateľ si tu môže vytvárať viaceré identity, pomocou ktorých sa prihlasuje do ostatných služieb. Dávame mu možnosť definovať nad dátami z identít závislosti a tým uľahčujeme správu a aktualizáciu dát. Dbáme na bezpečnosť a súkromie a kladieme veľký dôraz na používateľovu kontrolu nad celým systémom. Navrhnutý rozšírený protokol tvorí spolu so systémom na správu identít kompletný používateľský balík na administráciu svojej digitálnej identity z jedného miesta. Celý návrh sme úspešne implementovali do komunitného portálu matfyz.sk. Implementácia je dostupná na adrese `devprofile.matfyz.sk` spolu s demo webovou službou na adrese `demoapp1.matfyz.sk`. Zdrojové kódy su taktiež priložené na CD.

Kľúčové slová: digitálna identita, OpenID Connect, manažment identít, aktualizácia profilových dát

Abstract

Users on the Web are using extensive amount of services. In most of them they share information about themselves and create their user profile – their *digital identity*. One person nowadays owns dozens of such user accounts, which is often very difficult to administrate. Services barely communicate with each other and the user has to manage each of their accounts separately. Existing systems solve these problems only partially. The aim of this thesis was to design and implement a new system which provides users with an easy way of managing their digital identity in one place. In the fist part of our thesis, we build on the existing internet authentication protocol OpenID Connect. We have designed an extension to it, which enables broad and effective communication among the involved web services. One of the key features is the ability of the user to define automatic update of profile data in the services they use. In the second part of the thesis we have designed a system for administration of user identities. It allows users to create multiple identities of their, with which they log in to other services. We give users an option to define dependencies over the data in the identities and thus facilitate their management and update. We pay attention to security and privacy and emphasize the user control over the whole system. The proposed protocol extension with the system for identity management together form a complex user package for management of digital identity in one place. We have successfully implemented the whole proposal into the community portal matfyz.sk. The implementation is available at `devprofile.matfyz.sk` together with a demo web service at `demoapp1.matfyz.sk`. The source code for both is also accessible on the attached CD.

Keywords: digital identity, OpenID Connect, identity management, profile data update

Contents

Introduction

The Web consists of numerous services which can be effectively used only when the user creates a user account. The user usually fills out a form with their personal information like name or email address and creates a password to be able to log-in to the same account next time. Nowadays it is common that dozens of such accounts are owned by one person across many services they use. Since the services are usually completely independent from each other, the user needs to administrate all their accounts separately. If they, for example, moved to another city, they need to update their location on all the services manually. Sometimes this separation of services is useful as the user wants to keep some accounts isolated from each other - we do not want our business colleagues to recognize our "gaming profile" on the social network, where people who are interested in video games gather together. On the other hand, sometimes users would like their profiles at multiple web services to be recognized and linked together to form one *digital identity* of the individual. A musician might want to be recognized at various music-oriented social networks as the same individual. With many accounts also many other issues arise: having to remember many passwords, the need to re-enter the same data all over again at each new service we register to etc.

There already are some systems, frameworks and protocols that address some of the above mentioned issues with the user identity. Some of them focus on delegation of authentication to one place, some propose a way how to add pieces of user-generated content from one service to another, some reduce the number of passwords we use. However, none of the inspected options gives a satisfiable solution for all the issues. In particular, there is no system which facilitates the user administration and manipulation of their *digital identity* in one place and also no solution exists for the automatic updating of user information without the user having to manually perform the update.

That is why we have decided to design and implement a protocol, which would facilitate the use-cases, where the exchange of user-related information among multiple services is needed. In our design, for example, the relocated user can set up a policy to let the system automatically update their location on all the web-services on their behalf. Since these kinds of data are very sensitive, the user needs to have full control over which are exchanged and how. In our proposal, the user always stays in charge of all the data flows. This thesis also proposes a design of an identity management system,

where the user administrates their different identities (e.g. the gaming and professional identity from the first paragraph) and uses the system for logging in to other web services. In our system, we facilitate the separation of identities when desired as well as linking of identities when requested by the user.

To prove the usability and usefulness of our concepts we have fully implemented both the identity management system and the protocol at the `devprofile.matfyz.sk`. A demo web-service is available at `demoapp1.matfyz.sk`. The source code for both can also be found on the attached CD.

In Chapter 1, we define and explain the requirements, give an overview of the existing systems and analyse their contribution to solving the problems of the user web identities. In Chapter 2, we explain the details of the protocols and frameworks which are the foundation for our proposed solution; that is thoroughly described in Chapter 3, where we introduce our protocol for exchange of user data among services as well as the identity management system. In Chapter 4, we present our proof-of-concept implementation including its data model. Finally, the summary of our work can be found in the Conclusion.

Chapter 1

Background

1.1 Digital identity

As we already stated in the Introduction, users create user accounts all over the Web. The general terminology for these scenarios is coined in the ISO/IEC 24760-1:2011 standard [18] as follows:

entity: item inside or outside an information and communication technology system, such as a person, an organization etc.

identity: set of attributes related to an entity

attribute: characteristic or property of an entity that can be used to describe its state, appearance, or other aspects

So the end-user of the system is an entity which acts in the system using its identity, which is defined by the set of attributes. One entity can have multiple identities, which is common on the web — one user has dozens of user account across various domains. The user identity at the service is usually presented on the profile page of the user at that service, visible either publicly or only to a restricted group of other users of the service. In this thesis we will thus be using the words identity and profile as synonyms.

If the user wishes to keep some identities separated, it is usually achieved by selecting different values for the attributes that form the identity on that particular service. People use their real names when it comes to business related issues but tend to act under nicknames when it comes to more casual or hobby environment.

1.1.1 Requirements

With many identities at many services several problems arise. We sum up the high-level requirements on these systems into following points:

passwords: Usually username in comnination with password is the method used for authentication. Users have to remember excessive number of these combinations leading to password reuse, which leads to security issues.

re-entering of attributes: When creating new account, user has to go through the tedious process of filling out the attributes, although, they can be easily taken from another existing account.

updating of attributes: When certain attribute changes, it is probable that it changes for multiple identities, where the same attribute is present, and thus needs to be edited manually on all the services if the user wants to keep them up-to-date.

undesired separation: Services do not communicate with each other. The identities are completely separated from each other from the service's point of view. User might manually add links to their other profile pages in order for the visitors to be able to link their identities together. However, the services itself usually do not communicate.

desired separation: When user wants to have certain identities isolated it should be so. Colluding websites must not be able to reveal the links between identities when undesired.

whole digital identity in one place: Sometimes it is desired to present our digital identity in one place. Motivation might be, for example, when the user wants to build their online CV. They might want to include references from their colleagues and job history from their LinkedIn profile, their coding projects from their repository at GitHub, completed courses from e-learning platforms etc. A common standard for encoding these information is needed here in order to provide seamless mixing of information from many profiles.

user control of their information: User needs to have control over which profile data are aggregated, which are shown publicly, which information should be communicated about among the services and so on. The management of the control should be in one place to make it comfortable and transparent for the user.

1.2 Existing systems and solutions

Some of these problems are already addressed by existing systems. We will now introduce them briefly.

1.2.1 Password managers

The password problem is to some extent handled by password managers — special programs that take care of user passwords. They usually store them in an encrypted database which is accessed with the user's master password. The user thus only needs to memorize one strong password and all the other passwords are remembered for them. Some password managers store the data locally on the user's device, some use cloud-based storage space. A detailed definition, requirements and comparison of popular web-based password managers can be found in [21].

Password managers however do not solve any other issues from Section 1.1.1 (desired separation issue is satisfied since the accounts stay separated). These programs are suitable for people who do not want their profiles to be connected in any way and want to manage them completely separately and on their own.

1.2.2 Federated Identity

Another approach to solving the password problem is federated identity. In this approach multiple applications cooperate and form a system of mutual trust with an authorization server. When the user wants to log in to some application, authorization server confirms their identity by exchanging a token with the application. Since the two are in a trusted relationship, application relies on the authorization server and logs the user in. The involved applications can also exchange attributes of the user identity with an authorization server where this identity is primarily stored — this solves also the problem with the re-entering of attributes. Since the user has a stored identity with which they log in to all the applications, they do not need to take care of the attributes — they are copied for them.

There are several implementations of this principle ranging from open-source software, free software packages or mere protocols to be implemented by the members. For comparison of available solutions see our earlier work [30]. However, all the available solutions do not address the problem of profile isolations. They offer the user the possibility to use *one* profile for all the applications. This is good for example in a corporate setting, where the employees need to maintain the same identity across all the systems they use, but does not fit well into more general environment described earlier. The profile isolation can be bypassed in this setting by creating multiple accounts at the authorization server. This, however, brings multiple passwords, possibly same data are stored multiple times with no additional value added from the fact that they are the same, and the management is inconvenient, which were the things we wanted to avoid in the beginning.

The federated identity idea and principles are, however, very elaborate and address many issues related to identity management. One of the broadly used internet standards for federated identity on the internet is OAuth 2.0 and its next evolution OpenID

Connect. The biggest advantages of these are that they are interoperable, thus allowing for cooperation among any two services online. They operate over HTTP so there is no need for the two cooperating applications to be on the same network or operate over the same domain. More thorough comparison of OAuth to other options can be found in our earlier work [30], where we explain the reasons we decided to work with OAuth rather then other federated identity solutions. Since we will be using the OAuth later on in this thesis, we will now introduce the protocol.

1.2.2.1 OAuth 2.0

OAuth 2.0 is an authorization framework that arranges for a secure way of limited access to HTTP resources by third-parties [12]. We will demonstrate the whole protocol purpose and usage on an example of picture manipulating services. We have a web site that enables user to store their pictures in one place and administrate them by creating albums etc. Another website offers a collage maker, where user can use their photos to create new works. These two services are unrelated and running on different servers. The obvious way for a user to create a collage is to download the desired pictures from one site and then re-upload them to the other one for manipulation. In order to facilitate this cumbersome operation, the collage-making service might offer the user a possibility to give them their user credentials to the photo-depository service (usually username and password combination) so that it will perform the download/upload directly on their behalf. However, this approach is not very secure. The user is not only revealing all the pictures they have stored on the depository website but also giving the collage-service full access and thus control over their account. It may even lock them out of their own account if it wanted to - by changing the user's password. Better way to go would be to give the collage-service only a limited access to the resources it needs and nothing more. The OAuth protocol was designed to solve exactly this situation. More detailed description of the protocol entities and flows can be found in Section 2.1.

With OAuth the third-party can access some resource in a limited way and securely operate on it by receiving a special string — access token. Which resources are available and how to operate on them, however, depends on their provider. It is up to the provider what level of communication it supports with the engaged service. The OAuth 2.0 only specifies the way to grant access to these resources and not what they should look like. This mechanism is used for federated identity: identity provider defines a resource, where information about the user can be fetched and then grants an access to this resource to the other parties upon user approval.

1.2.3 Proprietary APIs and OpenID Connect

The communication among services (see requirements in Section 1.1.1) is nowadays mostly done by using proprietary APIs built on the top of the authorization protocol

OAuth 2.0. Big service providers each offer their own API through which other application developers can interact with their users. To name a few: Google has its *Identity platform* [11] with many APIs for their services, Facebook offers *Facebook Login* [7] with the *Graph API*, LinkedIn and their *Sign In with LinkedIn* [23], GitHub offers their service and API as well[10] etc.

The technique that all these identity providers are deploying when using OAuth 2.0 for providing authentication can be simplified with an analogy to a "valet key" [26]. In the parking lot with valet parking service you can give the attendant not your normal car key but a special one, which won't allow the car to be driven for more than limited number of kilometres etc. The same is true for an access token. In order to use this principle not for authorization but for authentication you could give the third-party this kind of a "valet key" to some property of yours and by visiting this property the third-party can believe that you are indeed who you claim to be. However, it is not secure to give out valet keys to your belongings to random third-parties. Better way would be to give them access only to some locker with a certificate confirming your identity so that they can verify your identity without having a possibility to cause any damage.

The implementation of this principle depends on the identity provider. It can issue access tokens that allows the application to access some basic profile information with a unique identifier of the user thus enabling to use it for logging users in. Cooperating application can entirely rely on it to log their users in — helping users to have less passwords to remember. But not only user authentication and federated log-in is supported by the APIs; some providers support access to certain resources without knowing the user's identity at all. For example the application can request access only to the profile picture of a user (they must give consent to it) and they are returned this image upon request, but the application does not know which user the picture is related to. The information exchange thus depends *solely* on the API of the particular provider.

When we compare some of the APIs that support user login (API versions at the time of writing), we see that they do the same thing in a similar way, but not the same. In the Table 1.1 we can see the ways to obtain a name of the authenticated user. Having received a grant (an access token) to access this user property, the application needs to make a call to *endpoint* and then search for a *property* attribute in the answer to get the user name. The last *update* column states whether it is possible to send a request to update this property at the provider.

In the table we can see that even in such a basic example as a name there are differences in the APIs. In more elaborate functionalities, the contrasts are naturally much bigger, which is partly understandable, since the providers offer different kind of services and available actions differ from each other (from GitHub one can ask to

provider	endpoint	property	update
Facebook	/me	name	NO
GitHub	/user	name	YES
LinkedIn	/people	firstName, lastName	NO

Table 1.1: Requesiting name of the user via various proprietary APIs.

retrieve the Git repositories, from Facebook their Likes and so on). However, the user profile related information are similar across all the platforms, yet not accessed in an interoperable way. For application developers this means that if they want to cooperate with multiple providers, they need to implement each API completely separately.

1.2.3.1 OpenID Connect — a unifying non-proprietary protocol

A call for a common way of executing user authentication (as mentioned, the proprietary approach is inconvenient for developers) led into an extension of the OAuth 2.0 standard. In order to perform the authentication in an interoperable and secure way, a new common open non-proprietary standard was introduced — OpenID Connect. This standard unifies the previously mentioned methods of accessing identity-related user data so that all the providers can offer the information in a common way. Identity providers that implement OpenID Connect can ask for certification of conformance with the standard. The list of the foundation certified implementations is accessible at `http://openid.net/certification/`. To name a few implementers: Google, Ping Identity, Deutsche Telekom, PayPal, etc. We can see that it is already supported by many major identity providers, thus it is very probable that this standard will be gaining more popularity in the future. In this thesis we will be building on the top of this standard and thus more information about it is in Section 2.2.

In OpenID Connect as well as in most of the inspected proprietary APIs, the flow of the data is only from provider to the application when it comes to identity attributes. The applications consume the data the identity providers are sending upon request. When an update to a profile attribute occurs at the identity provider, the application finds out next time it fetches the endpoint with user information and notices the difference. So the user information is flowing in the polling fashion rather then being pushed to the consumer.

1.2.4 Web hooks

In order to keep information up-to-date in the applications, some identity providers supports the use of Web hooks. This term was coined in the blogpost by Jeff Lindsay [22] back in 2007. Web hook is a simple HTTP callback which is triggered when certain event occurs. A server that supports Web hooks let its users to define callback

URLs. When the agreed event happens, HTTP POST request is sent to the defined URL containing information about the event. In our scenario the user would be the application and the server the identity provider. When an update request is unsuccessful the provider should retry more times over the following couple of hours. Many identity providers support the use of Web hooks, however, each provider's implementation is different and inconsistent with others (same as with proprietary APIs). Applications need to communicate with each identity provider separately.

As already stated, most of the data flow of the profile information is from identity provider to application. Although some of the providers do support the update of information to a certain level (see Table 1.1), it is usually limited to a small number of attributes that can be modified by an external application. In the current setting, the way permissions are asked for is that the third-party initiates what permissions it wants to get and then user confirms or cancels the approval of these permissions. In most of the systems, the default for all the requested permissions is "agree" and the user must actively de-select the permissions they do not want to give (*opt-out fashion*).

Thus, it could be dangerous to let the application ask for a permission to edit user's profile, since the user might mistakenly (by "only" confirming the default values) give it the permission to do so. It might cause them trouble, when potentially malicious third-party changes their profile at the identity provider in an undesired way. That is why it is understandable that in the current setting identity providers do not want to give the third-parties the possibility to ask for such permission. However, if we let the user initiate giving this permission only to the applications they trust, in an *opt-in fashion*, it might facilitate their identity management on the Web in general.

Profile attributes are not the only information that should be kept up-to-date. When the user decides to revoke the permissions given for certain application (or decides to delete their account completely), the application should be notified about the event in order to be able to adapt the things in their environment to the new situation. Some providers do support this feature (e.g. Facebook allows developers to register a Web hook for this event), but not all of them do and the level of support varies from provider to provider. Apart from account deletion, there are other security-related data that should be exchanged in order to keep users more secure. When a user account at the identity provider is corrupted, with this one account the attacker might get access to many other connected profiles of the user. The exchange of these kind of data might bring more security to user accounts used in federated identity fashion. A working group at OpenID Foundation is working on defining a standard for the exchange of such data (Risk and Incident Sharing and Coordination Working Group [29]).

1.2.5 Web ontologies

Returning to the requirements from the beginning of this chapter, another issue we are facing is how to put the individual pieces of digital identity together in one place (*whole digital identity in one place* requirement from Section 1.1.1). There are various existing implementations of how to showcase events and data originating from another source elsewhere. An example are LinkedIn certificates, where web-sites that issue certificates for their users (e.g some e-learning platform) can post this achievement to the user profile at LinkedIn, where it is subsequently displayed. This post, of course, needs to have the form defined by the provider in order to be understood and accepted.

1.2.5.1 RDF

The problem we are facing here is not specific only to this case. The World Wide Web in its current form is mostly meant for humans as consumers of its contents rather than for computers and their manipulation of the content. The machines can parse the pages in order to display them correctly and according to the prescribed layout; they can follow the provided links, but in general they do not have the idea of what the HTML pages represent. To go back to the certificate example — if the provider was offered a link to the description of it (intended to be read and used only by humans), it is not sufficient to understand what it is about. Based on this urge for a common language for representing information about resources on the Web, World Wide Web Consortium (W3C) came with a Resource Description Framework 1.0 (RDF) standard [24]. RDF gives us a syntax for making machine-readable statements about web resources. This syntax can be used to attach additional information to the web resource; the terms, which can be used to describe the resources are defined in web ontologies. Ontologies serve as "dictionaries", where the term's meaning and properties are explained. Thus, when a processing software knows the ontology, it understands the resources that are described with it. In our example, such ontology can define a general type certificate, where the properties such as who issued the certificate, how many points were achieved etc. could be present. The providers could then consume this data all in the same way.

1.2.5.2 Ontologies examples

An example of an ontology which defines vocabularies for actions invoked by the user is the Schema.org ontology [37] . It defines a type "Action" with many subtypes which is suitable for this scenario. An ontology created solely for description of user activities is ActivityStreams 2.0 [19] by Social Web Working Group [42] (W3C working group devoted to creation of the technical protocols, vocabularies and APIs for communication between the independent systems in social web environment in order to achieve the decentralized model of cooperation among heterogeneous elements).

However these ontologies are not yet deployed for exchange of user data between the collaborating services and mostly proprietary formats are used instead. In our thesis we want to propose an interoperable solution which can be adapted broadly a thus we have decided to use some existing ontology for the data exchange. From the two existing options we have chosen to focus on Schema.org, since it already has broader acceptance and usage (according to Schema.org some of its types are already used on more than million domains). However, the two can be translated between each other easily, so if a system understands one of them by providing equivalence relations it can be easily made to work with the other. The details of RDF and Schema.org ontology follow in the Section 2.3.

Chapter 2

Internet standards and protocols

In our design proposal in Chapter 3, we will be extending the existing internet standards and protocols and adding new functionality to them. In this chapter we will introduce the details of the standards and protocols that are relevant for the understanding of the design proposal.

2.1 OAuth 2.0

OAuth 2.0 [12] is an internet protocol used as a standard for delegated access to resources on behalf of another party. It is the second version of the OAuth framework [27] introduced by an IETF OAuth Working Group [16] in 2012, now widely deployed by many service providers. It is an authorization framework (the "auth" in the name thus stands for authorization and not authentication) which provides developers with an easy-to-implement way of access granting.

The protocol operates over HTTPS and serves as an additional authentication layer. We will explain the principles and entities of the OAuth protocol flow which we will need for our further work in more details. The full explanation is accessible in the specification [12].

The most important entities that interact in the protocol are following:

Resource owner (RO): Entity that can grant access to a certain resource. It is usually an end-user.

Authorization server (AS) : The server responsible for granting access to protected resources upon obtaining authorization from the RO (preceded by their authentication at authorization server). The resources are usually hosted on the same server and thus authorization server also acts as a resource server.

Client: Third party application, which manipulates protected resource on behalf of resource owner upon getting the authorization to do so. Cooperation with the particular AS must be established in advance by client registration at the AS.

Among other information, the RP registers its `client_id` — client identifier, `client_secret` — client password and `redirect_uri` — it's redirection endpoint are registered with the AS.

The protocol's principal building block is an access token. Authorization server issues access tokens (strings) to the clients on behalf of the resource owners. They serve as a certificate that the particular client has an authorization to access certain resource. The use of access tokens is expedient; they can be limited to have only partial access and we can invalidate them at any time thus hindering further undesired access.

For communication between client and Authorization server HTTP requests are used and following endpoints are engaged at AS:

Authorization endpoint: The authorization grant from the RO is obtained by client at this endpoint. Client redirects the user-agent here and the authorization server subsequently asks the RO for permission to grant access to the client.

Token endpoint: Exchange of authorization grants for tokens is performed here.

Clients need to implement only one endpoint:

Redirection endpoint: The AS redirects user-agent to this endpoint along with the authorization grant (confirmation that the user agreed to grant access to the particular resource for that client) or sends here an access token.

2.1.1 Authorization code flow

There are multiple grant types in the protocol, which can be used depending on the type of the client (whether it is standard web-based service, native/mobile app etc.). We will focus on the standard web-based services running in the browser, so we will be using Authorization code flow. Now, we will briefly explain it's principles and describe standard forms of requests and responses. Other grant types and their respective flows can be found in the specification [12].

The Authorization code flow consists of two round-trips from client to Authorization server. In the first one, client redirects the RO's user-agent to the AS (Authorization endpoint), where they are prompted to authenticate and to confirm the extent of the access that the client is asking for. Upon approval, the user-agent is redirected back to the client along with the authorization code (special token), which serves as a proof for the client, that they had received the approval (the RO authenticates only at the AS, so their credentials are not revealed to the client). The second part of the flow takes place in the background where servers communicate directly. The authorization code is exchanged for the actual access token at the AS's Token endpoint. The advantage of the background channel usage is that the access token is not exposed to the user

browser and thus its disclosure to the potentially malicious software with an access to the browser is avoided. Access token is valid for certain scope and time frame during which the protected resource can be accessed by client upon presenting it.

The request for an authentication code is done by constructing the URI to the Authorization endpoint of the AS in the *application/x-www-form-urlencoded* (HTTP GET method is used) [31] with following parameters:

`response_type`: With the value "code".

`client_id`: Identifier of the client as described above.

`redirect_uri`: Must match the URI registered at AS. It prevents sensitive information to be sent to false (e.g attacker's) endpoint.

`scope`: The requested scope. AS defines the scopes it supports in advance.

`state`: Value included for security purposes in order to hinder attacks (e.g. cross-site request forgery). This value is copied to the response by AS and client can verify that the response received is indeed response to its request. Otherwise attacker could send its access token to the endpoint rather than the victim's access token and eavesdrop the following manipulation (client would think that it is manipulating the victim's resources but it would not be).

Response to the authorization code request might be an error message, when the authorization code cannot be returned from various reasons. The response contains three parameters :

`error`: Possible values are self explanatory: `invalid_request`, `unauthorized_client`, `access_denied`, `unsupported_response_type`, `invalid_scope`, `server_error`, `temporarily_unavailable`, etc.

`error_description`: Human-readable additional message, describing the error.

`state`: As described above.

On the other hand, if the scope was granted by the RO and no other problem occurred the response contains only two parameters:

`code`: String to be exchanged for an access token, valid for a very short time.

`state`: As described above.

To exchange the authorization code for an access token a request to the Token endpoint is needed. For this request POST method is used and parameters are sent using Form serialization. Client can authenticate itself either by standard HTTP Authentication as defined in [9], or by including `client_id` and `client_secret` into the body of the request.

`grant_type`: With the value "authorization_code".

code: The authorization code obtained from the AS.
redirect_uri: Described above.
state: Described above.

An error message is similar to the one described above, so we will describe only the successful response parameters:

access_token: String value issued by the AS, which is usually opaque to the client.
token_type: We will be using the "Bearer" token type, which is also recommended by the community. This token type serves as the "key" to the resource and no further verification of the entity that presented it is required upon usage of it. It is thus important that the access token is not revealed to a third-party.
expires_in: Expiration time of the token.
refresh_token: Special token used when new access token is desired after expiration of the previous one.
state: As described above.

2.2 OpenID Connect

OpenID Connect 1.0 is an identity layer built on the top of an OAuth 2.0 [34]. It uses the existing OAuth 2.0 protocol flows and extends its requests in order to introduce a common standard for authentication. It is thus compatible with plain OAuth and Authorization servers already supporting OAuth only need to implement the new features in order to become OpenID providers (OP).

OpenID Connect uses the principles of OAuth where limited access and use of HTTP resources is defined and adds the missing part of *how* to provide the identity-related information. We will describe these extensions in the following section. OpenID Connect also defines a standard dynamic ways for clients (also referred to as Relying parties (RP), since they rely on the OpenID providers to authenticate their users for them) to discover and register themselves at the provider. It is inevitable for the Relying party and OpenID provider to have an established relationship before the user authentication with that provider might take place. The RP can either pre-register itself in a way supported by the OP (e.g. developers console) or make use of the dynamic discovery [33] and registration [32], when supported. However, we will not be focusing on this part of the RP interaction so we will further suppose that the RP has already used either method to register and exchanged all the necessary information with the OP (e.g. client secret). More information can be found in the corresponding specifications.

In OpenID Connect there are the same endpoints as there are in the OAuth: Authorization and Token at the server and Redirection at the client side. In addition to these a new endpoint is introduced: **UserInfo Endpoint**, which serves as a protected

resource where information about user identity can be fetched upon presenting valid access token — the values returned depend on what the RPs requested and the user authorization.

2.2.1 ID Token

The fundamental component of the protocol is the ID Token. It is a structure for passing user identity-related information to the Relying parties. It is a JSON Web Token (JWT) [2], which is a URL-safe and compact way of transferring attribute-value pairs that is well suited for using in query parameters of an URI or in HTTP headers. JWT encapsulates JSON object that stores the transmitted information. This object is passed as a payload to the JSON Web Signature (JWS) structure and/or JSON Web Encryption (JWE) structure and thus can be signed and/or encrypted. The structure of the JWT as JWS consists of three parts (each *base64url* encoded) as follows: JSON Object Signing and Encryption (JOSE) Header (the type of the token — e.g. for JWT the value is "jwt" — and the signing algorithm are declared here), the actual JSON object with the attribute-value pairs and finally the computed signature from the previous two parts. These parts are concatenated together with a period "." as delimiters which gives us the final JWT representation.

ID Token is a JWT, which contains assertions about user Authentication designated for the client. We will more closely describe the required assertions (also called claims), that are always present in the token:

`iss`: The identifier of the issuer of this ID Token. URL using HTTPS scheme with no query or fragment components. This would usually be the OpenID Provider's URL.

`sub`: The locally unique identifier of the end-user within the issuer. It is a case sensitive string at most 255 ASCII characters long. The sub together with `iss` is the globally unique identifier of the end-user at the RP.

`aud`: The client id of the RP to whom is this ID Token issued. This prevents the replay attacks, when tokens intended for other clients could be misused and re-send.

`exp`: The expiration time of this token. After this time the token must be treated as invalid. Represented as the number of seconds from Thursday, 1 January 1970. All the other datetime values are represented in the same manner.

`iat`: The time when the token was issued.

and some optional claims we will be using later:

`auth_time`: The time when the user authentication took place at the OP.

`nonce`: Case sensitive string value that is used to mitigate replay attacks. Client includes this value in the request for an ID Token and OP then includes the unchanged value into the token so that the client can verify that the request and response belong together.

ID Token also usually contains further claims. The exact structure depends on the information client requested to get from the OP.

The ID Token signature using JWS is required by the OpenID Connect standard and optionally it can be encrypted using JWE. Tokens thereby offer authentication, integrity, non-repudiation, and optionally also confidentiality. The big advantage of the ID Tokens is that they are stand-alone and can thus be used for example instead of a session cookies in the browser. When client logs the user in using OpenID Connect, they can store the ID Token in the cookie and verify its validity and expiration instead of storing session IDs on the server and authenticating users by comparing them.

2.2.2 Protocol flow

Three are three supported flows of authentication:

Authorization Code flow The most secure, used by the standard web-applications where direct communication from server to server is possible and the tokens can be exchanged without being exposed to the user. Two round-trips are needed and client authentication is possible.

Implicit Flow Suitable for apps running solely in browser (e.g. JavaScript based web-applications) where the tokens are returned directly without second round-trip. However, a security risk is present.

Hybrid Flow Mixture of the previous two where some tokens are returned from the authorization endpoint and some from the token endpoint.

As already mentioned with OAuth, in our thesis we will be working with the Authorization Code flow only a thus we will only describe this flow in more detail. The other flows can be found in the specification [34]. To keep it clear, we will now go through the flow and explain the differences from the OAuth form of requests and responses in its Authorization Code grant type.

The Authentication request is the OpenID Connect counterpart to the Authorization request in OAuth. The main difference that tells them apart is the scope parameter which is now required and in this case is set to `openid`. Scope parameter can also include other values which determines the user data they want to receive (e.g. `profile` is reserved for requesting profile claims). Other new optional parameters are introduced as well, through which RP can specify, for example, the language of the user interface

for the user, whether they would like the user to re-authenticate or re-approve the consent they gave them previously etc. Some of these additional optional parameters, which are required to be implemented by all OPs at least at a minimal level are:

`nonce`: To mitigate replay attacks.

`max_age`: Maximum time in seconds which the client is willing to accept to have elapsed from the last active authentication of the user at the OP. If the time passed is greater that this value, OP must try to re-authenticate the user.

`prompt`: Using this parameter, client can specify what type of interaction it requests the OP to establish with the end-user. There are four defined values:

1. **`none`:** The OP must not display any authentication or consent screen to the user and will thus continue only if the user is already authenticated and has a predefined consent for this client. Otherwise the response would result in an error message with the code `login_required` or `interaction_required`.
2. **`login`:** The OP should actively re-authenticate the user. If it cannot be done, error message (`login_required`) is returned.
3. **`consent`:** The OP should ask user to re-approve the consent given for this client; if consent is not given error message (`consent_required`) is returned.
4. **`select_account`:** The OP should prompt the user to select an account for this interaction. This is meant for users with multiple accounts at the OP for which they have active sessions, to choose which one to use; if it cannot be done an error message (`account_selection_required`) is returned.

The level of support of these parameters is optional for the OP and we will be dealing with some of them later in this section.

An example request (extra line breaks are added for better readability):

```
GET /authorize?response_type=code
  &scope=openid%20profile
  &client_id=sdsdf98d3
  &state=ftsa87asd
  &redirect_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb
 HTTP/1.1
Host: server.example.com
```

Upon receiving a request, it is verified in the OAuth fashion and furthermore the presence of other necessary parameters is verified and checked according to the specification (e.g. `openid` scope value). If the request is validated, OP authenticates the user (if the user is not already authenticated or the client requested special behaviour with the `prompt` parameter) and requests user authorization for the client-requested

information if not previously established. The response can then be either Successful Authentication Response, which looks the same as in OAuth 2.0 Successful Authorization Response, or an error response, where several new error codes are introduced, some of which were mentioned in the description of the `prompt` attribute.

The Token Request is the same as in the OAuth specification. Client presents the received authorization code and exchanges it for tokens. Upon successful validation (same as before, with an exception that the scope of the authorization code is checked and so determined that it is an OpenID Connect request), not only access token, but also ID Token of the user are returned in the OpenID Connect setting. Successful response thus contains an additional parameter `id_token`. The ID Token itself includes the parameters described above and optionally additional one: `at_hash`, which is a the hashed value of the access token string (the same hash algorithm is used as in the JOSE header of the ID Token). This enables the validation of the access token value. The ID Token validation requires more steps. RP must check all the parameters, whether they are present, if requested, and whether they have expected values (`iss`, `aud`, `nonce`, `exp` etc.). If the token was received via secure TLS channel, the server validation may be considered sufficient as the token verification, otherwise token signature needs to be validated.

2.2.3 Claims

In order to facilitate an exchange of assertions about user identity between RPs and OPs a set of predefined standard claims (identifiers of attributes the RP wishes to receive) is coined as well as a way for the involved parties to establish their own set of understood claims they want to exchange. To name a few of the standard ones (there are 20 defined claims and their meanings are mostly self-explanatory): `given_name`, `family_name`, `nickname`, `email`, `birthdate` and more. For the `address` claim its structure is defined by new JSON object, which is described in the specification. Predefined groups of these claims can be requested using the `scope` parameter set to `profile`, `email`, `phone` or `address`, or they can be requested individually using `claims` parameter.

We believe that the definition of these new standard claims is redundant, since there are web ontologies defining assertions about user identity (e.g. see Section 2.4.1). We will thus prefer to use properties defined by ontologies and make use of the OpenID Connect form of requesting non-standard claims via `claims` parameter.

The `claims` parameter is optional in Authentication request and if present it must contain a JSON object with two optional top-level members: `userinfo` and `id_token`, in which the requested claims from Userinfo Endpoint and claims requested to be present in an ID Token are enumerated. The value is either `null` (default request of the claim) or a JSON object with members: "`essential`" set to `true` — defining that this claim is necessary for a smooth course of authentication at the RP or "`value`"

("`values`") — defining that specific value(s) for the claim is requested (this is used, for example, when specific way of user authentication is desired from the OP etc.). Claims returned from the OP do not need to be only direct values but also so-called Aggregated and Distributed claims are supported. There might be other servers from OP that are storing claims about the user — Claim providers. These servers might cooperate with OP by sending signed JWTs with these claims to the OP, which then passes the JWTs further to the RP. This method is referred to as Aggregated claim. When the Claims provider does not send the JWTs, the OP cooperates with them by obtaining the access tokens and sending the claim identifiers together with the endpoint and access token for Claims provider to the RP, where it can be retrieved by the RP itself. The OP thus do not have a direct access to these claims but only mediates this information exchange.

2.3 Resource Description Framework 1.0

Resource Description Framework 1.0 (RDF) [24] standard is a common language for representing information about resources on the Web meant to be consumed by machines. It is now being revised to release the 1.1 version [38].

2.3.1 Datamodel

The basic concept of RDF is letting us make statements about Web resources. The structure of such a statement is very simple triple:

<subject> <predicate> <object>

copying the natural language, where the basic sentence also has the structure of subject, object and relationship between them. Here the subject and object refer to the particular resource and predicate (called property in RDF) expresses the relationship between them. This relationship is oriented — from subject to object.

The definition of triples is straight forward and general and thus requires further definition of how to identify subject, objects and properties (predicates). RDF 1.1 [38] proposes three options:

Internationalized Resource Identifiers: The identifiers of Web resources.

- IRIs are generalization of URIs, which allow also non-ASCII characters to be used (all Unicode characters are allowed).
- The usage for subjects and objects is intuitive. They can also identify the type of a relationship which can be reused in more triples. The RDF do not understand the meaning of a particular property though. It can be assigned a meaning by creating conventions and vocabularies explaining the meaning of a particular property — see Section 2.3.2.

Literals: Simple values, for which it would be redundant to create their own IRIs (dates, strings, numbers, etc.). These are then represented as literals associated with their RDF-defined datatype, in order to be interpreted correctly.

Blank nodes: The blank nodes are just an extension to the basic schema of IRIs and literals, since the two form the foundation, which is sufficient to express RDF statements. In the situation when there is need to make a statement containing an object or subject, whose global IRI is not important, we can get the use of blank nodes as a kind of variables. These can be than used in the place of both subject and object. However, blank nodes are strictly local to one file or scope and are not part of the RDF abstract syntax. They are dependent on the concrete implementation and syntax.

2.3.2 Ontology

The introduced RDF scheme gives apparatus for naming and interconnecting resources. The problem is that two different entities can refer to the same thing using two different IRIs. As an example we can take two databases of high schools. In both of them, there is a column for the principal's name. We can specify by RDF which column it is (<column 4><represents><principal>). As specified above, each element is represented as IRI and thus everyone (including the machine) can find them on the Web. However, the second database might use the RDF to identify the column as *headmaster*. The two identifiers obviously refer to the same concept, but the machine which wants to combine the two databases does not know about it. To solve this problem a common collection of information is needed, to avoid such inconsistencies.

Ontology is a term used for a document which define terms and relationships between the terms. It was adopted by Web researches to describe common vocabularies which contain the taxonomy and inference rules [1]. Taxonomy defines the classes to which objects may belong and the relations between them (e.g. *Headmaster* is a subclass of the *Human* class). Inference rules define what can be deduced from the facts — for example rule can say that *high school* is the type of secondary school and if certain school is of the type *high school* it can be deduced that that school is a secondary school.

Thus the meanings of the terms used can be explained by linking to the used ontology. Two databases using the same ontology can then be processed by the computer by looking up the terms in that ontology. However, when we look back at the principal/headmaster example, the problem might not be solved, if the two point at two different ontologies (one using principal and the other headmaster paradigm). It is therefore required that the two ontologies should provide equivalence relations, which explain that the two terms actually mean the same thing.

In a nutshell, ontologies are vocabularies that give semantic meanings to the used syntax. It is essential for the usage of RDF that common standard ontologies are adopted on the Web. Without them everybody would define their own set of terms and relationships used in their part of the Web, thus preventing the whole idea of linked data to be used across all the domains, unless these ontologies are describing their translations to the other ones.

2.3.3 RDF Schema

To enable the definition of new ontologies, RDF includes apparatus to support the creation of such new vocabularies. RDF schema [3] is a semantic extension of the RDF vocabulary. It is a language which specifies how the semantics of RDF data is defined. The basic structures are:

Classes: RDF uses the paradigm of classes known from Object Oriented Programming. Classes are resources and therefore are also identified by IRIs. The two most important classes in RDF Schema are (syntax for the class is in the so-called prefix form, which we will describe in more detail in Section 2.3.4):

- **Resource:** The super class of everything in RDF. All RDF resources are the instances of this class.
- **Class (*rdfs:Class*):** The super class of all resources, that are RDF classes.
- **Property (*rdf:Property*):** The super class of all the resources, which are RDF properties (see below).

properties They define the relationships between resources (subjects and objects). The most important ones are:

- **type (*rdf:type*):** States that a certain resource is an instance of a certain class
- **subClassOf (*rdfs:subClassOf*):** Defines the hierarchy of the classes. A class can be subclass of another class and inherit its characteristics.
- **subPropertyOf (*rdfs:subPropertyOf*):** Defines the hierarchy of properties.
- **domain (*rdfs:domain*):** Enables the creation of constraints on usage of some property. This property specifies which classes are allowed to be used as a subject in a triple with the particular property resource.
- **range (*rdfs:range*):** Put a constraint on objects of a particular property.
- **comment (*rdfs:comment*):** Human-readable description of the property.

2.3.4 RDF formats

Until now we have introduced the abstract RDF data model, but we have not talked about the concrete syntactic format of the RDF. There are multiple serialization formats: RDF/XML, JSON+LD, RDFa, Microdata etc. According to Web Data Commons project statistics from 2014 [41] the most widely used formats are the last two. That is why we decided to focus on the RDFa and Microdata in our thesis. We will now introduce these two in more detail.

2.3.4.1 RDFa

RDFa (Resource Description Framework in Attributes) [13] is a syntax, which extends the HTML-family of languages, and is thus easily implementable into existing HTML web-sites. RDFa introduces a way of embedding the RDF structures into HTML.

RDFa introduces new attributes, which do not collide with traditional HTML attributes; if the site is enriched by RDFa attributes, its visual representation thus remains the same as before.

By default the subject of the RDF triple is the IRI of the current document. However, multiple resources may appear on one page — the `resource` attribute is used to specify what the treated subject is. The inside of the element with the `resource` attribute is then considered to belong to this subject. Another way of declaring the subject is to use `about` attribute — the advantage is that it can be placed into the same element with the `property` attribute, thus making the syntax more clear. Another attribute concerning the subject part of the RDF triple is `typeof`. It specifies the type of the subject (the class of which it is an instance); usage of `typeof` creates a blank node which is described further by predicates enclosed by the element with the `typeof` attribute.

The predicate is determined by appending `property` attribute to the element containing the object of the triple with the IRI of the used predicate as a value. To avoid the cumbersome repetition of the namespace used within more `property` attributes, the `vocab` attribute is used in the parent element with the value of the namespace URL. When referencing some property inside this parent element, the address is taken automatically from the `vocab` and the `property` value is concatenated to form the valid identifier of the predicate.

Objects of RDF triples are either the content of an element with the `property` attribute, or it might be listed as a value of a `content` attribute within the same element (`href` attribute is also automatically treated as an object of the predicate). If the object is a literal, additional `datatype` attribute to indicate the type of the data is enclosed.

We will demonstrate the enrichment of a Web site with RDFa on the following example, where we state that Mária Šormanová studies at the Faculty of Mathematics, Physics and Informatics:

```
<body vocab='http://example_vocabulary.org/namespace/'>
 <div typeof='Student'>
  <p> My name is
   <span property='Name'>Mária Šormanová</span>
   and I study at
   <a property='isAStudentOf' href='http://fmph.uniba.sk'>
    Faculty of Mathematics, Physics and Informatics
   </a>
   It is a faculty of
   <span about='http://fmph.uniba.sk'>
    <a property='isAFacultyOf' href='http://www.uniba.sk'>
     Commenius University
    </a>
   </span> in Bratislava.
  </p>
 </div>
<body>
```

2.3.4.2 Microdata

Another way of encoding the machine-readable data into HTML documents is by using Microdata standard designed by Web Hypertext Application Technology Working Group [14]. Similarly to RDFa, it adds special meta data to an HTML code of the website. The syntax extends the HTML5 language.

The building blocks of the Microdata are groups of name/value pairs. Groups are called items and are created by an `itemscope` attribute. All descendant elements (with some Microdata attributes) are then treated as belonging to this item. However, if also some elements which are not direct descendant of this element need to belong to the item, a list of their IDs in a `itemref` attribute can be added and the Microdata parser will crawl them as well. The name/value pairs,called properties, are created by adding `itemprop` attribute to a descending element of an item. The value of the attribute specifies the name of the property and the content specifies the value. When the value is supposed to be a hyperlink it is given by the `href` attribute of an `a` element (analogously the picture URL is given by `src` in an `img` element).

To use Microdata in context of existing vocabularies, `itemtype` attribute is used to indicate the type of an item. It is added to the same element as `itemscope` and its value is the identifier (URL) of the item of the vocabulary.

An example of Microdata embedded to the HTML (equivalent to the one we presented in RDFa):

```
<div itemscope
 itemtype='http://example_vocabulary.org/namespace/Student'>
 My name is
 <span itemprop='Name'>Mária Šormanová</span>
 and I study at
 <a itemprop='isAStudentOf' href='http://fmph.uniba.sk'>
  <span itemscope itemref='faculty'>
   Faculty of Mathematics, Physics and Informatics
  </span>
 </a>
 It is a faculty of
 <div id='faculty'>
  <a itemprop='isAFacultyOf' href='http://www.uniba.sk'>
   Commenius University
  </a> in Bratislava.
 </div>
</div>
```

2.4 Examples of common vocabularies

The real usefulness of all the yet mentioned models comes only when the pages which are marked-up with them use some common ontologies. That gives third-party applications the chance to understand them. The more the particular ontology is being reused the more powerful it becomes. We will now introduce the vocabulary, which we decided to use in our thesis — Schema.org. It is already widely deployed (according to the homepage of the vocabulary [36] more then million domains use some of the Schema.org types) and is gaining more popularity. Another advantage of Schema.org is that it can be used with RDFa and Microdata in a convenient way.

2.4.1 Schema.org

The Schema.org [36] is a general-purpose vocabulary designed in the cooperation of major search engine companies — Google, Bing, Yahoo! and Yandex. The aim is to make the schemas, which can be adopted and implemented by web developers to mark-up the web pages and thus provide search crawlers with structured data. Since any application can use this free format, not only search engine can benefit from the defined schemes.

Schema.org introduces vocabulary types (RDF classes) and their associated properties. Some types are more specific — they inherit the properties of broader (parent) types. Multiple inheritance is supported. The broadest type is a `Thing` with properties: `name`, `description`, `url`, `alternateName`, `sameAs`, `image`, etc. All the other types

inherit these from the `Thing`, except for basic data types, which have their own root type: `DataType` with descendants: `Boolean`, `Date`, `DateTime`, `Number`, `Text` and `Time`.

Schema.org tries to provide the vocabulary for a very broad variety of items with the ambition to provide sufficient apparatus to categorize most of the data on the Web. The complete list of all types can be found in the full hierarchy of Schema.org [35]. For our purposes the types `Action` and `Person` are important. We will use the `Action` type for annotating user activities on the web services where they have a registered account and create content. This type defines broad range of properties which can be associated with an item. The `Person` type with its properties will be used for addressing user profile data.

Although Schema.org provides really rich structure, it surely does not cover everything out there. For this situations it can be either combined with some other vocabulary, or a completely new scheme for some items might be designed and used (if they prove to be useful they can possibly be eventually adopted by Schema.org), or Schema.org can be extended by its extension mechanism — the "/" character is concatenated to the type one wishes to extend and followed by the name of the new type.

Chapter 3

Proposed solution

We have decided to build our work on the top of existing internet protocols and standards, introduced in Chapter 2, to make it compatible with the systems that already support them. Our foundation thus be the OpenID Connect. The backward compatibility will be maintained in our design and so the systems interested in supporting our extensions only need to add the new features to their existing infrastructures.

This chapter introduces the whole specification we designed, which extends flows of the OpenID Connect as well as the design of an identity provider supporting the identity management functionality. The requirements notation and conventions are the same as defined in the Section 1.1 of the OpenID Connect specification [34].

In both of our designs we propose the concept of "attribute dependency". We allow the user to define dependencies over the attributes of their identities; it does not matter whether the attributes are present at the identity provider or at the cooperating web-services. This approach facilitates the maintaining of all the user information up-to-date.

3.1 Exchange of updates to identity attributes

3.1.1 Endpoints

We are introducing two new endpoints to the OpenID Connect (OIDC) protocol:

UserUpdate endpoint An OP's endpoint, where the incoming update requests from RPs are coming. It is an OAuth 2.0 protected resource.

Update endpoint An RP's endpoint. It needs to be registered in advance at the OP in the same manner as the other URLs (redirection, homepage, etc.). At this endpoint the RP must be listening for requests from OP with the information about the changes in user identity — see below.

3.1.2 Protocol flow

We will explain the additions to the protocol in the Authorization Code flow, as we did in Section 2.2.

The Authentication request has the same structure as the OpenID Connect Authentication request. In order to differentiate them from each other at the OP, we introduce a new scope keyword `update`. It is used in addition to the `openid` scope, thus complying to the OIDC protocol and letting the server know that the RP is supporting our extension.

3.1.2.1 Web ontology defined claims

Although OIDC scope values (`profile`, `email`, `address`, `phone`) can be used to request the defined claim sets in accordance with OIDC protocol, we recommend the usage of the `claims` parameter to ask for claims specifically. The set of standard claims registered at Internet Assigned Numbers Authority (IANA) [17] contains only a small number of basic claims. Additional claims are allowed by the specification, with emphasis on creating collision-resistant names. The standard usage of the additional claims is that the OP declares the names and definitions of the claims it supports. The RP can choose from these claims and decide which ones to ask for. The understanding of the meaning, however, must be done in advance. In our solution we want the attributes to be as flexible as possible and we want to support new attributes without previous understanding of them from both sides. That is why we propose the usage of web ontologies for requesting claims. There already are ontologies defining terms for assertions about user identity (e.g. see Section 2.4), which are suitable for our purposes. Moreover, if an RP wanted to ask for more specific claim, not defined by any ontology yet, it can either define it's own or extend the existing one. The only requirement is that the ontology must be defined using RDF Schema language.

The RP asks for the claim defined by some ontology in the same way as for non-standard claim. The OP then fetches the received ontology URL and parses the returned content to search for the description and range (data type) of the property it is dealing with (`rdfs:comment` and `rdfs:range` — see Section 2.3.3). If it cannot find the mentioned descriptions, it handles the URL in a standard way (it either is a previously established identifier and treated accordingly or just the identifier without further description is presented to the user to treat is as they please). By automatically obtaining this information it can provide the user with the full description of the property the RP is asking for. The user can then decide whether to enter and enclose the data to the RP. The great advantage of our web-ontology based approach is that the OP does not need to have any idea about the meaning of the data it is dealing with beforehand. The user understands it and decides on their own; the OP serves as a mediator only.

3.1.2.2 Files handling

Most of the attributes related to the identity are strings or some other data type which can be expressed as a string with format constraints (date, time, number, etc.). Another group of attributes are files. These are nowadays typically only profile pictures, but with our proposal any file attribute can be asked for and thus a new group of attributes arises (e.g. a PDF file with the user's CV). Files are passed around via providing the URL where the file can be fetched. However, a problem arises, since a URL string as a value of an attribute does not necessarily mean that the attribute is a file. It can also be a value for the "homePage" attribute, in which case we do not want to download any data from the URL and we just want to provide it as a value itself. To overcome any misunderstandings, we introduce a mandatory convention to end the URLs that are to be treated as values itself with an additional backslash ("/") and URLs to be treated as files with the file extension of the file retrievable at that URL. When a certain file is passed by sending the URL where it can be fetched, the sender is required to keep the URL active for at least 24 hours. This URL should include a randomly generated part with sufficient amount of entropy and be in no way linkable to the user by itself. This is to protect the privacy of the user and prevent guessing of URLs by a third-party.

3.1.2.3 User consent and claims policy establishment

Upon receiving and validating the new version of the Authorization Code request, the OP is obliged to authenticate the user (according to the OIDC specification) and to obtain a user authorization. If the Identity management is present at the OP, the user is now prompted to choose the identity they want to use for the RP (see Section 3.2). Then the requested claims are matched against the attributes present in the user identity at OP. If an attribute matching specific claim is not present, user must be prompted to either match some other attribute from the identity manually (e.g. the OP did not recognize that the "profileImage" and "profilePicture" refer to the same thing, but the user knows it and can put those two together) or add a brand new attribute matching the claim. If the claim is requested using ontology, the user is moreover presented with the description and data type of the claim (see previous section) to facilitate adding of a new attribute.

In addition to the consent for releasing the requested information to the relying party, the user must be prompted to choose (for every attribute independently) whether they want the changes to this attribute to be shared with the RP and whether they are asking the RP to inform OP about the changes to this attribute done at the RP. These two options are independent from each other, so the user can also consent to only one of them or neither. The default predefined options must be set to false so that the user actively has to opt-in for the exchange of any data upon updating. If the user chooses to update the attributes, they become *dependent* on each other, as we already

mentioned in the beginning of this chapter. The attribute at the RP can be *dependent* on the OP's attribute and vice versa.

The OP must store the user configuration of the policy in order to be able to verify requests and enable the alteration of the policy.

After obtaining the user consent and if no error was encountered a standard Successful Authentication Response is sent back containing the authorization code. Otherwise standard OAuth 2.0 Error response is sent back

3.1.2.4 Access Token and ID Token exchange

A Token request is sent from the RP to the OP with standard parameters from OIDC — among others the received authorization code. Upon validating the request, the OP prepares the response containing the standard parameters from OIDC including the ID Token with the values for claims the user has given consent to. If the user has chosen to establish an update policy (see Section 3.1.2.3) for at least one of the attributes, a new parameter `claims_policy` is present in the response. This parameter contains a JSON object with the configuration of the exchange policy user has created in the user consent step. Top level members of the object are the descriptors of the claims that the RP has asked for in the request and the user wants to establish an exchange policy on (claims without policy are recommended to be omitted in the policy description). The member values are one of the following:

`null`: If no policy was created for the claim. However, it is recommended to omit the claim with no policy completely from the JSON object.

JSON object: The policy setup object with following members:

`update_from_rp`: Indicates if the user wishes the RP to send the updates back to OP. The possible values are `true` or `false`, the default being the latter.

`update_to_rp`: Establishes the direction of exchange from the OP to RP. The values are the same as above.

Unlike OIDC, in our proposal we require the `sub` parameter to be pairwise unique for *every RP*, in order to prevent services to identify that they are dealing with the same user.

Let us assume that the RP has asked for claims: *http://schema.org/givenName*, *profilePicture* and *http://xmlns.com/foaf/0.1/title*. The user has asked the first attribute to be updated in both directions, the second one not updated at all and the last one only from OP to RP. The corresponding `claims_policy` parameter looks as follows:

```
{'http://schema.org/givenName': {
   'update_from_rp':true,
   'update_to_rp':true
 },
 'http://xmlns.com/foaf/0.1/title': {
  'update_to_rp':true
 }
}
```

The RP must store the policy configuration and is obliged to execute the required update operation when the corresponding update event occurs.

3.1.3 Update operation

The update operation is sensitive since it deals with the user data and the user might have many attributes depending on the change of the particular one. The malicious attempt to alter certain data without the user's knowledge might have unpleasant consequences. If a malicious request with an update of the profile picture contained a compromising photo, this update might propagate to other profiles of that user, which might cause discredit of the person. Thus security is important and above all integrity and confidentiality. That is why we decided to employ the existing techniques from OIDC, which can grant us these qualities and extend them for our purposes.

3.1.3.1 Update from RP to OP

When going through the successful Authentication flow of the OIDC protocol, the outcome is that the RP receives both ID Token with the user data as well as the access token to access the UserInfo endpoint. The changes to the attributes of the user profile at the RP ordinarily happens when the user is present (logged in). During the time the user is present, there is no obstacle for an RP to have an unexpired access token at its disposal (received from the Authentication flow or if one expires, the RP can re-initiate the authentication and obtain a new one). If the attributes at the RP can change when the user is not present, they can establish an offline access (see section 11 in the OIDC specification [34]) to retrieve an access token when the user is not present. Thus it is possible for an RP to always have an unexpired access token with which it can access the UserInfo endpoint. We decided to also make use of this access token to access the UserUpdate endpoint.

The UserUpdate endpoint is an OAuth 2.0 protected resource which is accessed using HTTPS protocol and a valid access token, thus satisfying both integrity and confidentiality. The UserUpdate request is an HTTP POST request sent to this endpoint with parameters encoded either using Form Serialization, or JSON Serialization. The authentication with the "Bearer" access token is either by using HTTP Authentication

header [9] or by including the parameter `access_token` into the request. The remaining two compulsory parameters in the request are:

`sub`: The identifier of the user (obtained from the ID Token), whose attributes are about to be updated.

`claims_update`: Contains a JSON object. The members are the identifiers of the claims, which has been updated with their new values as the member values.

An example of a request for update of the *http://schema.org/jobTitle* for a user with an `sub` identifier "aso8inf5" using JSON Serialization:

```
{'sub': 'aso8inf5',
 'claims_update': {
  'http://schema.org/jobTitle':'junior programmer'
 }
}
```

Upon receiving the UserUpdate Request the OP verifies the validity of the access token for the user with the `sub` identifier. If verification is unsuccessful the Error response is sent as defined in section UserInfo Error Response in OIDC specification [34]. Otherwise, the OP checks whether the user has given consent to update this claim from RP and if true, updates the attribute and sends the HTTP response with the "OK" status code to indicate that the request was successful. If the OP server is temporarily unreachable, the RP is obliged to follow the retry policy from section 3.1.7.

3.1.3.2 Update from OP to RP

For an update from OP we need a way to guarantee that the incoming update request is indeed coming from the OP. ID Token is a structure in OIDC that allows us to pass user attributes and sign it to guarantee integrity. That is why we have chosen to use ID Tokens to pass the updates of user data. However, since the RP has not initiated the sending of this token, we cannot use the `nonce` to mitigate the attacks based on the reuse of a token. Thus it is recommended to set only a very short validity of the token. At most 5 minutes expiration time is recommended, in order to give the RP enough time to process the request and to mitigate replay attacks at the same time.

We are introducing a new ID Token claim `claims_update` for passing updated attributes with the same content as described in previous section in UserUpdate Request. The ID Token is sent to the Update endpoint at the RP in an HTTP POST request with a parameter `id_token` (Form Serialization) containing the token. If this endpoint does not support HTTPS, the ID Token must be encrypted (using the client secret of the RP) to ensure confidentiality. The successful response and retry policy are the same as in the previous section. The validation of the ID Token is the same

as in the OIDC. If the RP is using ID Tokens e.g. as a supplement for having to store user sessions, or it has passed tokens to other components of the service, etc. (see Section 2.2.1), in the time of receiving the token with the update information, the RP is recommended to request fresh ID Token with updated information. This fresh token should then be distributed to the other components.

Upon receiving the token with updates the RP is obliged to update all the requested attributes in the user identity immediately.

3.1.4 Claims policy edit, change consent

OP must provide the user with a way to alter their consent about the attributes released to the RP as well as about the claims policy. OP can use the same dialog as in the Section 3.1.2.3 to let the user edit their decisions (also the previously denied claims must be now offered to be optionally accepted). The OP notifies the RP about these changes via ID Token sent to the Update endpoint (all the requirements already introduced hold) with the new JWT claim `claims_policy_update`. Its structure is the same as for `claims_policy` JWT claim. However, the policy setup object can now contain additional members:

`deleted`: The possible values are `true` or `false`, which is the default. Indicates that the user no longer wishes to share this attribute with the RP. Upon receiving the `true` value for this member, the RP may no longer use this attribute and must delete it from the user profile on their side immediately.

`added`: The user decided to share this attribute with the RP in spite of their previous decision not to disclose it. The values are same as for `deleted`. If set to `true`, member `value` must be present as well.

`value`: The current value of the attribute to be shared with the RP. The policy established on the other claims in the past and not mentioned in this ID Token remains the same. The RP must update the stored policy and act in accordance with the new rules.

3.1.5 Account deletion

If the user decides to delete their account at the OP or completely revoke access for an RP, the RP needs to be notified about the event. We again use the ID Token and introduce a new JWT claim `ace` (account event) with keyword value `DELETED`. This claim is designed for future use with further keywords to communicate about other events with the account. This is, however, currently out of the scope of this thesis. Upon receiving the token, the RP should delete the user account according to their Terms Of Service. If the user only wants the connection between OP and RP to be

erased and does not want the account to be deleted at the RP, the RP should offer the user a way to migrate the account to another OP or to a local account at their service. This migration needs to be done *before* the delete action is taken at the OP and the token with the delete event is sent.

An example payload of an ID Token with the `ace` claim is (notice the expiration time — the difference between `iat` and `exp` is 300 seconds, which is the above mentioned upper limit of 5 minutes for validity):

```
{'iss': 'https://matfyz.sk',
 'aud': 'appsclientid',
 'iat':1460190120,
 'exp':1460190420,
 'sub': 'aso8inf5',
 'ace': 'DELETED'
}
```

3.1.6 Migrating account

If the user already has an established account at the RP with identity attributes and wishes to connect it with the OP, we propose a new Migration Request. This request is an extension of the Authentication request, described earlier in our proposal. A new parameter `claims_migration` is added containing a JSON Object with the top level members being descriptors of the claims present in the user's identity at the RP with their corresponding values. RP might ask the user, upon choosing to migrate the account, to select which attributes should be sent to the OP. The `claims` parameter remains the same as before thus allowing the RP to ask additional attributes from the OP not present at the RP yet. When such request is received, OP validates it in the standard fashion and then prompts the user to merge the values received with the corresponding attributes at the OP - user chooses which value to keep for each attribute from the two options (RP's and OP's). If the Identity Management is present (Section 3.2), the user can also be offered an option to create a new identity with the received attributes and values. Upon merging the data, a standard Authorization Code flow continues as described earlier.

3.1.7 Retry policy

If the endpoint to which a request is being sent is unreachable, or some server-side error occurs at the endpoint, which we can detect from a status code other than "OK" in the response, the entity sending the request is obliged to follow the retry policy. The policy binds the entity to try to send the request at least 8 more times over the following 72

hours and, if needed, then try one last time at least 72 hours after the original request was sent. The implementation of exponential backoff in frequency [1] is recommended.

3.2 Identity management

Now we will propose a system where the user can administrate multiple identities of theirs in one place. We present a solution which solves the problem of the user presentation on multiple web services. Users do not want to access all the services they use with only one identity. They need to keep certain services separated and thus the identities under which they act on these services need to be different. The following description is a framework for the identity systems that wish to support identity management.

To facilitate the management of the different identities we present a system where the user has all their identities under control and in one place. They can manage these identities and decide how they interact and how they are displayed to the public. From outside (from the third parties' point of view) the system is completely opaque and looks as an ordinary identity provider. This means that the Relying party, when authenticating users via this provider, has no idea about the identity management that might be deployed there. From their point of view they interact with a standard user having one identity at the identity provider. From the user's point of view, they see all the identities they have created and they are given possibility to choose which identity to use with the particular Relying party.

3.2.0.1 Authorization flow

We add a new step into the Authorization flow — after redirection from the RP to the identity provider with an Authentication request, prior to showing the user the consent screen (described in Section 3.1), the user is asked to choose from their identities or is given an option to create a new one. If the option to create a new identity is chosen, the identifiers of the attributes of the identity are pre-populated with the claims asked by the RP. User can thus create a tailored identity exactly for use with the particular RP. This identity is then stored as any other identity at the provider and the user can potentially use it for other RPs as well. The decision which identity to use for the RP is stored and next time when the same RP sends the Authorization request, this step is omitted and RP communicates directly with the selected identity.

The number of identities the user can own is unlimited. The user gives each of them unique name in order to manipulate them more easily. This name, however, must not be accessible neither for the public nor for the RP.

[1] In exponential backoff the pauses are increasing by some multiplicative factor. This, for example, prevents overloading of a busy server.

3.2.1 Attributes

We put a strong accent on the flexibility of the user identities and their attributes. We want them to be as interoperable as possible. Nowadays, when creating a user profile at any service (including existing identity providers), user is presented with a form consisting of a number of attributes to be filled out. However, the attributes themselves cannot be altered. The only choice users have is whether to fill them out or not. In our system, we want the user to have the power over what kind of attributes they want to have present in their identity. E.g. if they wish to share their favourite colour they are free to do so by adding an attribute "myFavoriteColour" with the desired value. In this fashion, they can assemble the whole identity as they wish.

Attributes can be freely added and deleted from the identities as the user pleases (when an attribute is tight up with an RP, the RP is notified about the deletion — see Section 3.1.4). They are divided into two groups — file and non-file attributes. Non-file attributes can be of multiple types (text, string, date, number, email) but they all have in common that they can be stored and passed along as strings. The data type is mainly used for validation of the user input — all the dates must have the required form, all the numbers must consist only of digits etc. With file attributes, the user uploads the file to the system, where it is then stored. When passing such attribute to the RP a URL is generated at which the file is accessible, as described in Section 3.1.2.2. The implemenation is free to choose a set of supported file formats. However, it is recommended to support at least the following: JPG, JPEG, MPEG, PNG, PDF, MP3 and TXT.

Each attribute has it's main name identifier, by which it is addressed in the system. In addition to the main identifier, the user can define the main ontology-based identifier (these two can be the same). Attribute can also have any number of additional identifiers associated with it. This is to facilitate the claims matching (requested from the RP — see Section 3.1.2.3) but also when one value is sufficient for multiple attributes — e.g. city of birth is the same as the current city. In this case the user only needs to attach additional identifier to such attribute.

3.2.1.1 Dependencies on attributes

Although identities are different, they may have several attributes in common. In order to facilitate the update of attributes across more identities, we present the user with an option to define "dependencies" on attributes. Each attribute can *depend on* any number of other attributes in the identity management (non-file attributes can depend on non-file attributes and file on file). If any of the attributes on which the attribute depends is updated, this value propagates to the depending one. The relationship *depend on* thus forms unidirectional edges between attributes. This gives the user great flexibility in defining the flow of the data in their identities. Together with the

exchange of data with RPs (sec. 3.1) this creates a complex life-cycle of an update. With appropriate setup, the change of an attribute value at the RP might be sent to the identity provider, where the change propagates from one identity to another and if the other one is connected with another RP and a policy is defined as to update that attribute at the RP, this change propagates all the way to the second RP.

3.2.2 Visibility

All the identities have two modes of visibility: *public* and *private*. If the private mode is selected, the identity is not accessible by anyone except for the owning user. In public mode, a public URL is created for the identity, where the identity is presented as a public user profile. The URL, however, must be in no way linkable with other identities of the user. From public point of view it is simply a profile of one of the users of the service — no other connection can be revealed.

3.2.3 Operations on identities

We will now briefly describe the operations through which the user can manipulate and organize all their identities.

3.2.3.1 Create

Creation of a new identity consists of setting up the name for the identity, its visibility and adding initial attributes. The pre-defined suggestions for attributes might be provided by the implementation but are not necessary. The user is given a free hand in assembling the contents.

In order to facilitate the creation of an identity, the user should be given an option to use their existing profiles at other providers. If the other service was supporting our update extension to OpenID Connect (see Section 3.1), they could use the Migration request and create a new identity out of the migrated data passed by the RP. However, we wish to make use of the third parties, even when they do not support our extension.

For this purposes, we again want to utilize the web ontologies. Many services are already enriching the public profiles of their users by annotating the data with ontology-defined properties. In our system we suggest that the user enters the URL of their public profile at another service and the identity management system parses the HTML returned from that URL and looks for the data annotated with the person-related web ontology properties (e.g. *http://schema.org/Person*). It will then open the form for creation of a new identity and pre-define and pre-populate the attributes according to the properties found. User can now made additional changes to the identity and confirm its creation. If any file-attributes are present (e.g. a profile picture), it will download the file to the identity management, to be accessible even after the deletion

at the third party service. This exempts the user from the cumbersome procedure of downloading and re-uploading the picture themselves. The created identity is then stored and can be used as any other.

3.2.3.2 Delete

Deleting an identity results in deletion of all their attributes and their relationships to other attributes from different identities. If RPs are connected to this identity they are notified about the deletion (see Section 3.1.5). If this was a public identity, it is no longer accessible at the specified public URL.

3.2.3.3 Edit

Editing an identity enables users to change the values of the attributes and also to alter the attributes' definitions. Attributes can be freely added or deleted (when an attribute is associated with the RP, this RP must be notified as explained in Section 3.1.4). Also the change of the identity visibility and name must be offered to the user.

3.2.3.4 Merge

Two identities can be merged into one. Upon selecting this option, the system tries to match attributes of the identities with each other using their identifiers. For each pair of attributes (each from one identity) a set of common identifiers is created. If there exists an attribute from one identity for which more than one such set is *non-empty*, the identities cannot be merged and the user is prompted to resolve the conflicts. If there are no such conflicts, the user is presented with a form in which they are requested to choose which value should be kept for the matching attributes (the value from the first or from the second identity). All the other (not matched) attributes are simply copied to the final merged identity. If the two values for the matching attributes are distinct, upon selecting one and confirming the merge operation, all the depending fields of the other attribute need to be updated, as well as all the connected RPs.

Merge of two identities must be opaque for the connected RPs. For them it is merely seen as as an update to the attributes. Everything else remains the same — e.g. the `sub` identifier. All the RPs that were connected to either of the merging identities are now connected to the merged outcome. For each pair of merged attributes, the identifiers and the *depend on* relationships are merged as well; the merged field has all the dependencies from the previous two identities except for the dependencies between the two themselves.

3.2.4 Connected RPs

As already mentioned, each identity has its RPs that are connected to it. The list of these must be visible to the user and a way of deleting a certain RP and restricting its further access to the user data must be present. Also a link to the form with the policy update must be accessible from the identity.

3.2.5 Connected third parties

In Chapter 1, we defined a requirement for creation of a unified user profile, where user-created content from various profiles of theirs would be collected and presented in a common way. Now we will present our design of how to make this aggregation possible.

3.2.5.1 Format of user activities

Our aim is to make use of existing standards so that the developers have little additional work to make their systems meet our requirements. We have chosen to focus on *http://schema.org/Action* type and collect all the data annotated with this ontology to create a stream of user activities from their other profiles. The support of other ontologies is endorsed as well (e.g. Activity streams — see Section 1.2.5.2). This approach has an advantage that the activities all have the same attributes and thus we can display them in a similar fashion. An example (using microdata) of an annotated *http://schema.org/LikeAction* — notice the use of the `description` and `endTime`:

```
<div itemscope itemtype='http://schema.org/LikeAction'>
 <span itemprop='description'>
  <a href='http://example.com/mariasprofile'>Mária</a>
   liked a book called
  <a href='https://www.goodreads.com/book/show/3'>Harry Potter</a>.
 </span>
 <span itemprop='endTime'>2016-03-05T06:30:00</span>
 <span itemprop='agent'
  itemtype='http://schema.org/Person itemscope>
  <a href='http://example.com/mariasprofile'> Mária Šormanová </a>
 </span>likes
 <span itemprop='object'
  itemtype='http://schema.org/Book> itemscope>
  <a itemprop='name'
   href='https://www.goodreads.com/book/show/3'>
   Harry Potter</a>
 </span>
</div>
```

In order to aggregate the actions, we need the URLs of the user profile at the third parties where the annotated activities are located. For each identity in the identity management an option to connect third party must be present. User enters the URL of their profile at the third party. To verify, that it is indeed their profile the html `rel='me'` in `a` element is used. The use of this attribute is already supported by numerous services. It enables users to add links to other profiles of theirs and the service adds this attribute to the link thus demonstrating that this particular link leads to other profile pages of the same user. So if a user wants to add third party profile to their identity (the identity must be public), they need to first add the public link to this identity to the profile page, where, if supported, it will be annotated with the `rel='me'` attribute. If the user then enters the URL at the identity management, the system is able to verify the ownership of the profile by searching for this attribute and the value of the link. Next step after verification is the filter setup, where the user can choose which activity types they want to be collected and presented on their public profile home page at the identity provider (e.g. with Schema.org ontology they might choose only to display LikeAction and DislikeAction and nothing else). Upon establishing the filter, the setup is complete. In this fashion, the user has full control over their public profile contents.

The visitors on the public profile of the user can choose to show the user activities and then see them all sorted according to the `endTime`. The system parses all the connected profiles' contents for activities, they are then filtered according to the policy defined by the user, sorted, and finally displayed. The `description` is printed for each action, thus allowing visitors to follow the links provided by third parties to their content.

Chapter 4

Implementation

We have fully implemented the proposed solution and created a proof-of-concept demonstration of the proposed extensions and features. Our implementation is a part of the complex matfyz.sk portal, where user authentication is used to log users in to multiple sub-portals of the service.

4.0.5.2 Matfyz.sk portal

Matfyz.sk serves as a community portal for students and teachers of Faculty of Mathematics, Physics and Informatics of the Comenius University with an ambition to become an unofficial homepage of the faculty. The portal is being developed and maintained mostly by students as parts of their theses. More details about the portal can be found on the portal's homepage [25].

The portal consists of multiple sub-portals (profile.matfyz.sk, courses.matfyz.sk, blog.matfyz.sk, wiki.matfyz.sk, etc.). The user authentication is done via profile.matfyz.sk which serves as an OAuth 2.0 authorization server and the other sub-portals are cooperating with it as its relaying parties.

In our implementation, we have transformed the profile.matfyz.sk into an OpenID Provider, supporting all the proposed features from the Chapter 3. First we will describe the data model we will be using and then technologies used and the implementation itself.

4.1 Data model

For our implementation of the OpenID Connect basic flow, we have decided to use a PHP library oauth2-server-php (developed by Brent Shaffer — documentation available at the library homepage [39]). The library itself uses several tables with the `oauth_` prefix. In order not to interfere with the library's tables, we did not rename or delete any of their columns; we only added some new to them. We will mention the tables to

which we have added new columns and those that are important for understanding of the following sections. More information can be found in the library documentation.

`oauth_clients`: The table where information about Relying Parties is stored. We have added a new column `update_uri` for storing the URL of the new RP's endpoint - Update endpoint.

`oauth_users`: The table storing user credentials (username and password). Includes also `id_user` column which is the user's identifier across all the tables.

`oauth_authorization_codes`, `oauth_access_tokens`, etc. Tables storing the tokens and data related to each of them. We have added a new column `claims` for storing identifiers of the attributes, which were requested and granted for this token.

From now on, we will focus on the tables created by us.

4.1.1 Identity management related tables

All the tables directly related to the identity management have the prefix `identity_management_` and are shown in the Fig. 4.1. The foreign key constraints are represented by curved lines.

The `identity_management_identities` table stores all the identities created by users. The `id_user` defines the user, to whom the identity with the `id_identity` ID and `name` belongs. The `id_external` contains a 32 character long alphanumeric unique external identifier of an identity, which is used for generating public URLs (see Section 3.2.2). The `visibility` defines whether the identity is private or public. The `external` column is by default set to "0" — other possible values will be explained later.

The `identity_management_attributes` contains all the attributes in the identities, each having their uniqe ID (`id_attribute` column). For each attribute its main identifier (`display_name`) as well as main ontology-based identifier (`ontology_url`) are stored. The `type` column defines the data type of the attribute, which contains one of the following values: string, number, date, email, file. The `value` column stores the actual value of the attribute; values for all the data types except for *file* are stored as varchars. The actual format of the varchar depends on the data type (`type` column). For files, the `value` column stores the file name, including the filename extension stored in the filesystem. For how the files are stored and accessed see Section 4.2.

Each attribute can have multiple identifiers — not only `display_name` and `ontology_url`. All the additional identifiers are stored in the `identity_management_additional_names` table.

In our system, we need to create and store dependencies between attributes belonging to identities managed by our system as well as between the attributes at relying parties.

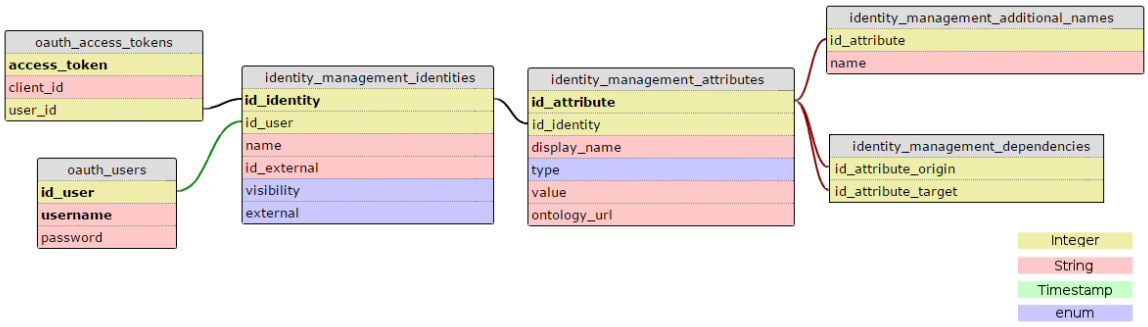

Figure 4.1: Data model of our implementation of the identity management system.

To make the dependency handling consistent and simple, we have decided to store a "pseudo-identity" called *client identity* every time a new RP is connected to the user and their consent is given for the RP. When user makes a decision to use particular identity with the RP a new client identity is created, assigned an ID and stored among other identities (this client identity expresses the relationship between user and that RP). Each requested claim becomes a client identity attribute, which are stored in the same way as attributes of normal (user-created) identities. How the matching between the identities and RP's client identity is stored is explained in Section 4.1.2.

Client identities will be stored in the `identity_management_identities` along with normal user-created identities. They have the `external` column set to "1" indicating that it is a client identity. Further only `id_user`, `id_identity` and `id_external` are filled out with all the remaining columns having the value `NULL`. The value of the `id_external` column will be used as a `sub` user identifier when communicating with this RP. This gives as a pair-wise unique identifier for a user — each RP is given a different `sub` identifier even though they are communicating with the same identity. In our scenario the authorization codes, access tokens etc. are assigned to the user identities which the user decides to use for the particular RP. Thus for each of the tokens we will assign the corresponding client identity to which it was issued. The foreign key from `id_identity` to `user_id` at `oauth_access_tokens` demonstrates this relationship. As we did not want to interfere with the library's tables we did not change the name of the column from `user_id` to `id_identity`, but we will use the `user_id` column for storing IDs of identities instead of storing user IDs.

Each requested claim by the RP becomes a *client attribute* associated with the respective client identity. These client attributes are stored in the `identity_management_attributes` and are assigned `id_attribute` as normal user-created attributes. Client attribute's `display_name` is the identifier of the requested claim (e.g. "http://schema.org/givenName"). If the identifier is ontology-based then it is stored in the `ontology_url` column as well. Other columns are not used by the client attributes and have the value `NULL`.

In the `identity_management_dependencies`, all the relations between attributes.

The direction of the relationship is that the target attribute (`id_attribute_target` column) is dependent on the origin attribute (`id_attribute_origin` column). The user's wish to update certain fields on update to or from RP (claims policy defined in Section 3.1.2.3) is stored in this table as well. The client identity attribute is made dependent on the attribute from the identity, which is used for that RP and vice versa respectively.

4.1.2 Exchange of updates related

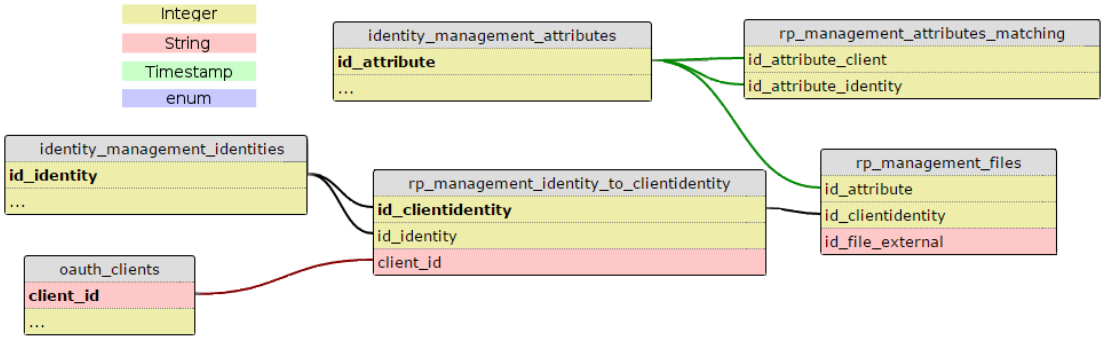

Figure 4.2: Data model containing the tables that are related to the relying party management.

The tables related to the exchange of updates and storing the user setup for the Relying parties have the prefix `rp_management_` and are shown in the Fig. 4.2.

For each RP and user pair we need to remember whether the user has already given consent to the RP or not and if so, which identity of theirs the user wishes to use with this RP. This information is stored in the `rp_management_identity_to_clientidentity` table. A row in the table means that the user wishes to use identity (`id_identity`) for the particular RP (`client_id`) and that a new client identity was created for this relation which has the `id_clientidentity` ID. If for the given RP there is no row in the table with any of the identities related to the user, then user has not established any relationship with this RP yet. If this RP initiates the authentication flow for this user, the user will be prompted to select the identity they wish to use for this RP during the flow.

During the very first Authentication flow with the RP, when the user chooses an identity to use with the RP, the requested claims are paired up with the attributes from the user identity. This mapping is stored in the `rp_management_fields_matching` table. If client attribute from the client identity (`id_attribute_client` column) is paired up with one of the attributes from the user identity (`id_attribute_identity` column) and if the user gives consent to release this attribute to the RP, this relationship is stored in the table. If no attribute is paired up with this client attribute or if the

user did not give a permission to reveal the value of that attribute to the RP then -1 is stored in the `id_field_profile` column.

When passing files to the RPs an opaque URL address is created, where the file can be fetched by the RP. One such URL for each RP that has access to the particular file is needed to avoid colluding websites to identify the user by these file URLs. In the `rp_management_files` table for an attribute with "file" type (`id_attribute`) and particular client (`id_clientidentity`) an `id_file_external` is stored which is a component used for generating the final URL to be passed to the RP.

4.1.3 Aggregation of content related

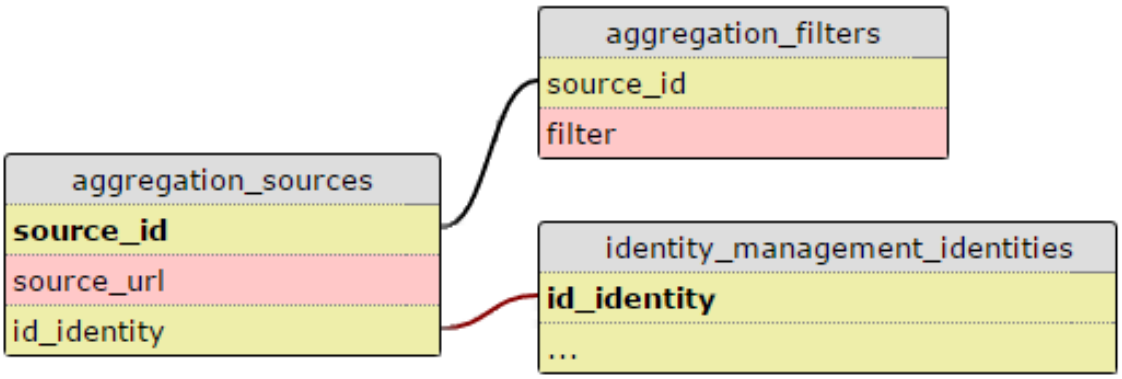

Figure 4.3: Data model of aggregation of user data from other sources

In the Fig. 4.3 the two tables for storing user setup of data aggregation are depicted. The `aggregation_sources` table maps the source URL's (the third parties from which the user wishes to aggregate their content) to the identity with which the user wishes this source to be used used. The `aggregation_filters` table stores all the identifiers of elements (e.g. http://Schema.org/Action subclasses) that the user wishes to aggregate from that source.

4.2 The implemented system at profile.matfyz.sk

4.2.1 Technologies

The profile.matfyz.sk subportal is a complex web application which was developed using various technologies as a part of our Bachelor's Thesis [30]. To seamlessly add new functionality without rewriting the current code, we decided to continue with the already deployed frameworks and programming languages, which we introduce in the

next section. We are also using several new PHP libraries, which are described in the Section 4.2.1.3

4.2.1.1 PHP, MySQL, CSS and JavaScript

A standard combination of web programming languages was initially chosen for this project. We use PHP, an open-source scripting language, for the interactive parts of the website, whose execution is done on the server, so the user does not see the code producing the website results. MySQL is an open-source relational database system suitable for open-source projects, which we use as a database solution. We further use CSS Cascading style sheet language for the appearance and design of the sites (the original design done by Roman Janajev in his Master's Thesis [20] — we will be further extending the design). And for interactive parts of the site run in browser (suitable for dynamic interaction with the user in web forms etc.) we use JavaScript. Reference and manuals for all of the mentioned languages is accessible at W3Schools [40].

4.2.1.2 CodeIgniter

To facilitate the development in PHP, many frameworks are available which provide libraries for common tasks such as form generation, URL resolving, etc. A framework called CodeIgniter [6] is used for profile.matfyz.sk. The basic structure follows the MVC (Model-View-Controller) design patter. The CodeIgniter controllers are accessed via URI; the name of the controller is written as the first segment of the resource path and the function to be called within the controller is indicated by the second segment — if no second segment is present then the `index` function is called.

4.2.1.3 PHP Libraries

For implementation of the requested features several new PHP libraries were needed to fulfil the task. We are using open source libraries, all available at GitHub. To name a few (the rest can be found in the source code of the application): **Guzzle** [5] — a PHP HTTP client containing adapters for sending and receiving requests, **oauth2-server-php** [39] — OAuth 2.0 protocol flow helpers, **password_compat** [8] — for secure passwords hashing, **jwt** [28] — creation of JWTs, **EasyRdf** [15] — consuming and producing RDF syntax, **rel-me** [4] — rel="me" attribute manipulation.

4.2.2 Controllers

Since controllers are the crucial elements that interconnect users with the whole system, we will demonstrate the implementation by introducing its controllers — views and models will be briefly described along. We are only listing the controllers that we have added or completely redesigned (comparing to the original state of the profile.matfyz.sk).

As we have stated in Chapter 1, we are using words identity and profile as synonyms. We decided to use the term profile in the implementation, since it is more common to refer to user's identity at certain web-service as their "user profile". Thus we believe that this terminology is more user-friendly and we will be referring to identity as profile in the rest of this chapter.

First, we will describe the controllers handling authentication flows and exchange of information between identity provider and relying parties and then the identity (profile) management controllers.

4.2.2.1 Authorize controller

Authorize controller resides at the Authorization endpoint and takes care of the Authorization requests. It validates the input, extracts the information from the requests parameters to determine what kind of a request it is. Depending on the presence of the `prompt` parameter and its value it authenticates or re-authenticates the user (redirects them to Login controller if nobody is logged in) and fetches the profile to be used with this RP (redirects to Choose profile prompt controller if none was previously selected). Upon successful completion of all the steps it issues an authorization code and redirects back to the client. The communication with the database is done via `token_model` model, which contains functions for manipulation of tokens and other OpenID Connect related data.

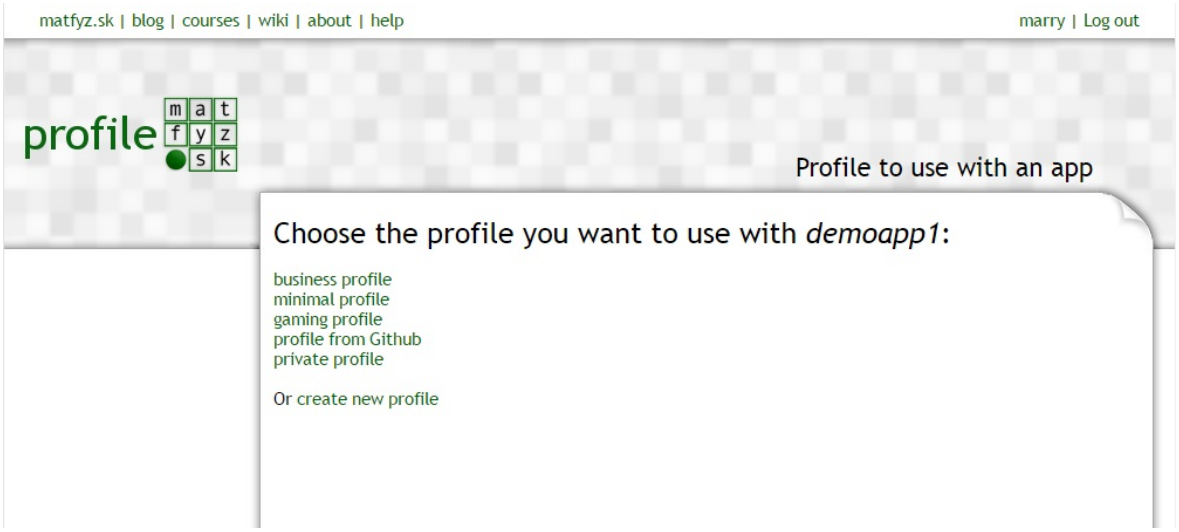

Figure 4.4: The profile selection for use with the RP.

4.2.2.2 Choose profile prompt controller

The Authorization controller redirects to the Choose profile prompt controller to prompt the user to choose the profile they want to have associated with the particular RP. The Choose profile prompt controller prompts the user to choose from the list of their

profiles and an option to create a brand new profile is offered as well. This prompt is showen in the Fig. 4.4.

When the user chooses the profile they want to use, with the help of the `profile_management_model`, the controller matches the requested claims with the attributes from the profile. The user is then shown a consent screen with two categories of claims: claims matched to some attributes from profile and unmatched claims. For each matched claim, the user is presented with three checkboxes: update to RP (TO), update from RP (FROM) and consent with the release of the attribute value (OK). Each unmatched claim has the button to match an attribute from the profile manually ("paper clip" icon) or to add a completely new attribute to the profile ("plus" icon). If the claim is requested by ontology identifier, the description of the claim (fetched from the URI of the claim identifier) is provided as well. The consent screen is shown in the Fig. 4.5.

Figure 4.5: The consent screen.

After the user creation and confirmation of the desired setup is the whole configuration stored into the database. The user is then redirected back to the Authorize controller, where the flow finishes and the user is sent back to the RP with an authorization code.

4.2.2.3 Token controller

The Token controller resides at the Token endpoint. It is responsible for an exchange of an authorization code for the access and ID token. Upon receiving the request, the

authorization code is validated and then the claims associated with it are included into the ID Token. The claims policy is included in the response together with an access and IT tokens. As the `sub` identifier of the user is used the `id_external` of the client profile of this RP (`id_external` is used instead of the `id_identity` in order to efface the connections which might be suspected when the successive values of IDs were used). Any time the `sub` parameter is presented from an external source (from the RP) it needs to first be translated into the internal identifier for the associated client profile (`id_identity`).

The facilitation of the database communication is done via the `profile_client_model`, where the queries related to the management of the RPs are located and the `token_model`.

This controller is designated for server to server interaction only and has no associated views with it.

4.2.2.4 Userinfo controller

The Userinfo controller implements the functionality of the UserInfo endpoint. Upon receiving an access token it validates it and returns the agreed claims, in the form of an ID Token, from the profile associated with the RP to which the access token belongs.

This controller is also for server to server use only.

4.2.2.5 UserUpdate controller

The implementation of the UserUpdate endpoint is done by the UserUpdate controller. Again, only server to server communication is supported. Received request is validated and all the required parameters are checked if they are present. Error messages are sent if the request does not have the prescribed form. Upon receiving the valid update request, the controller is responsible for updating the corresponding client profile attribute's value in the database (for files also direct download from the provided URL is needed) and checks for any depending fields.

All the dependencies are then updated within the local profiles of the user at the OP with the help of the `profile_management_model`. This, however, might yield also updates to other RPs, connected to these profiles, so ID Tokens need to be issued and sent to the Update endpoints of these RPs. This process is executed via `external_update_model`. This model batches all the updates designated to one RP into one token and sends all the created tokens to the recipients (RPs).

Upon successful completion of the whole process an HTTP OK response is sent back to the client.

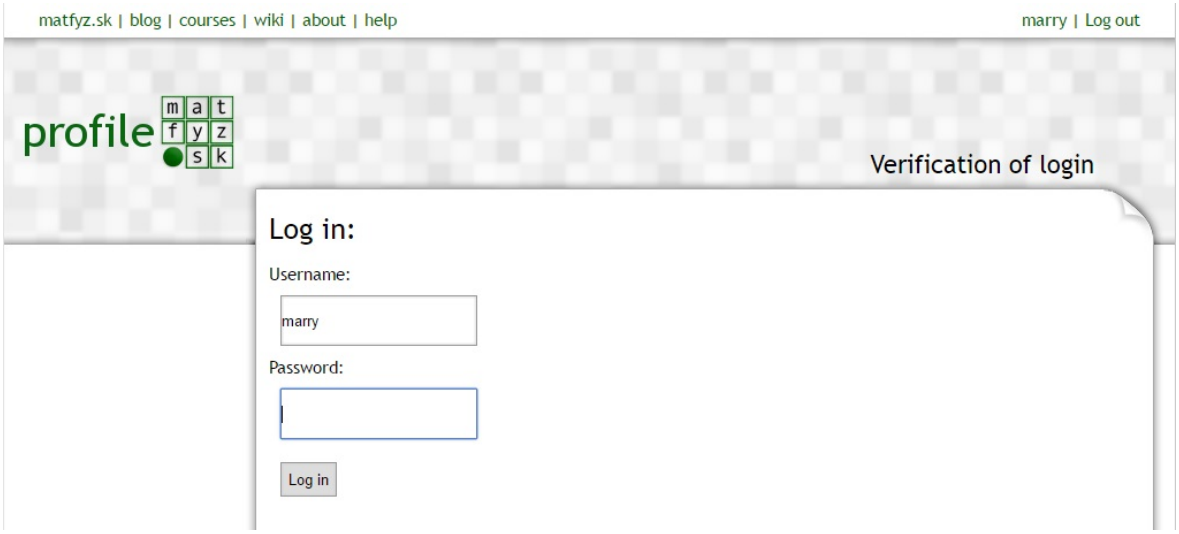

Figure 4.6: The re-login screen.

4.2.2.6 Login controller and Password recovery controller

Most of the functionality of the Login controller is left from the original implementation. It checks whether somebody is logged in and if not a login form is presented. The `membership_model` serves for validating user credentials. We have added the re-authentication function called by putting "relogin" as a second segment of the URI when calling the Login controller. This function checks that somebody is already logged in and subsequently prompts them to re-login. If successful, it redirects back to the URL set in the cookie by the Authorize controller (that is the only controller that might want the user to re-login). The screen for the re-validation of the logged-in user is in the Fig. 4.6.

We have also added the "forgotten password" function which generates and sends the password recovery links to the user's e-mail address (the user needs to enter the address given at the registration time as verification).

The Password recovery controller validates the recovery link and prompts the user to create a new password. Upon entering and confirming the new password, it stores the hash of the new password in the database and lets the user log in with the new credentials.

4.2.2.7 Profile management controller

The Profile management controller is the main controller responsible for all the operations related to the identity management (throughout the rest of the chapter we will use the term "profile management"). This controller is accessible only for the logged in users. The screen is divided into two parts — menu on the left and the main view in the center of the screen. From the menu the user can choose whether they want to see the list of all the profiles they have in the system together with their attributes and

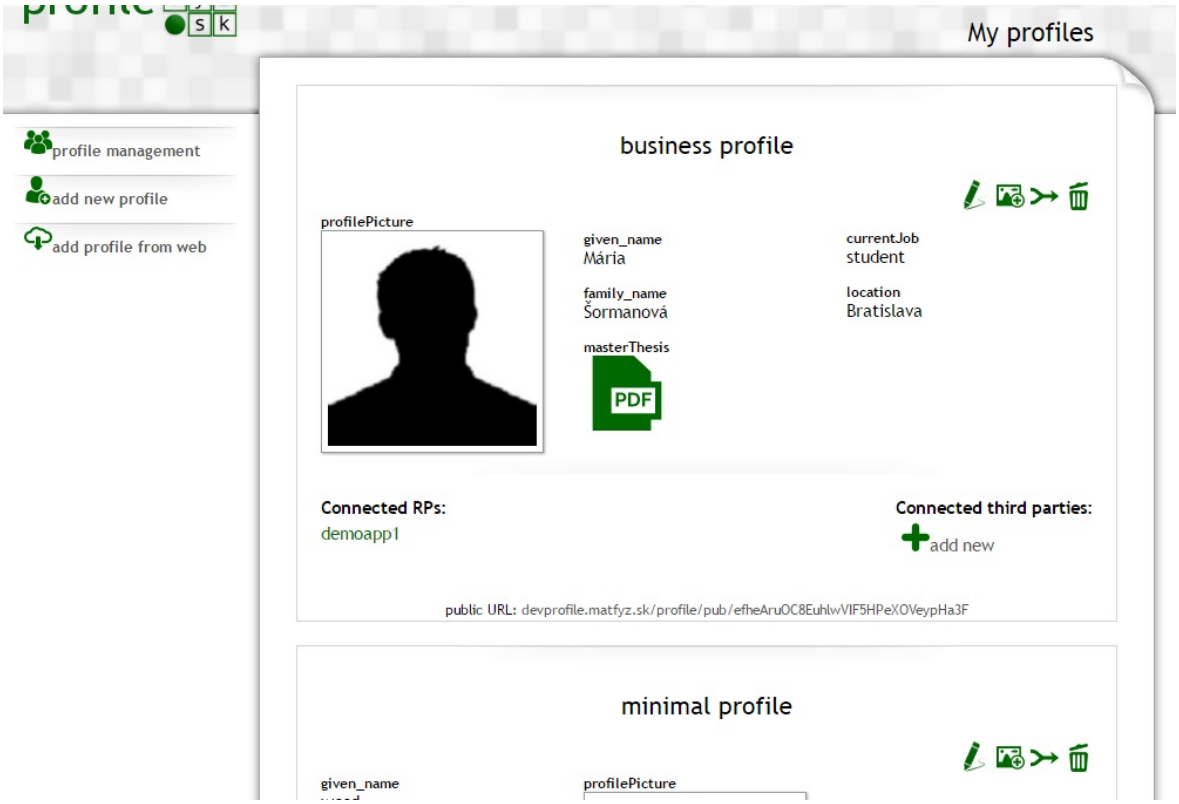

Figure 4.7: The list of the user profiles.

control buttons ("profile management"), whether they want to add new profile to their account ("add new profile") or an option to fetch the profile attributes from an existing profile available on the Web is offered to the user as the third option("add profile from web").

A part of the list of all the profiles is depicted in the Fig. 4.7. Each profile is enclosed in a card with grey border, which are arranged under each other. One such profile card consists of the name of the profile, the list of all the attributes with their values, the list of the connected RPs, the list of the connected third parties and the control buttons (edit profile and its non-file attributes, add new file, merge profile, delete profile).

Non-file attributes can be edited by clicking on the "pencil" icon in the profile card. The edit profile form with all the fields is presented to the user, where they can modify all the existing values, ontology identifiers or names of the attributes, as well as add completely new attributes to the profile. The dependencies on attributes are created in the edit profile form as well.

File attributes can be edited by clicking on their picture, upon which a form is shown. This form contains an option to upload a new version of the file as well all the other attribute manipulation options. New files are added by clicking on the "picture" icon in the profile card. Each profile has two designated folders in the file system: one identified by the `id_identity` and one by `id_external` of the profile. Files are stored in the former and symlinks to them with an opaque identifiers are stored in the latter.

If the profile is public, this second folder is used for presenting files in the public view of the profile so that the `id_identity` of the profile is not present in the file's URL. The same symlinks principle is used, when communicating with RPs. Each client identity has its own folder, identified by its `id_external` (`sub` of the user used with this RP), where opaque symlinks are created, so that the RP cannot deduce anything from the received URL.

If the profile is set to be public, the public URL of the profile is visible in the bottom of the card. This URL leads to the Profile controller.

Upon clicking on one of the connected RPs, the user is brought to the RP management controller, where they can edit the established policy with the RP.

When clicking on one of the third parties, the Third party management controller is called, where the user can edit the filters established for that third-party. Upon clicking on the "add new" link, the same controller is called to present the user with the form for adding new third-party.

All the operations on the profiles are executed with the help of the `profile_management_model` which has numerous functions that communicate with the database and facilitate the manipulation of the profiles and attributes.

4.2.2.8 Profile controller

The Profile controller is responsible for displaying the public profiles. It only has two functions, which are called from the menu on the left: `pub` and `activities`. The first one takes the third URL segment and searches for a profile in the database which has the `id_external` equal to the value of the segment and is set as public by the user. If no such profile exists, the controller prints out the message that no such profile exists. If the profile is available, then its attributes are printed out on the screen in a single profile card.

The `activities` function is responsible for the aggregation of data from all the third-parties that the user has set up with that profile. It fetches the URLs of all the third-parties and searches for the *http://schema.org/Action* subtypes that the user has set up to be aggregated from them. All the fetched data is then sorted according to the `endTime` property and printed out as a feed.

4.2.2.9 RP management controller

This controller allows the user to edit the policy they have set up with the RP or delete the connection with the RP completely. It assembles a form consisting of all the claims that the RP has asked an access to, which are divided into three categories: the claims for which the user has given their consent, the claims that can be matched to some of the existing attribute from the profile and claims that does not match to any attributes from the profile. The user can edit their consent or the update flows. Also

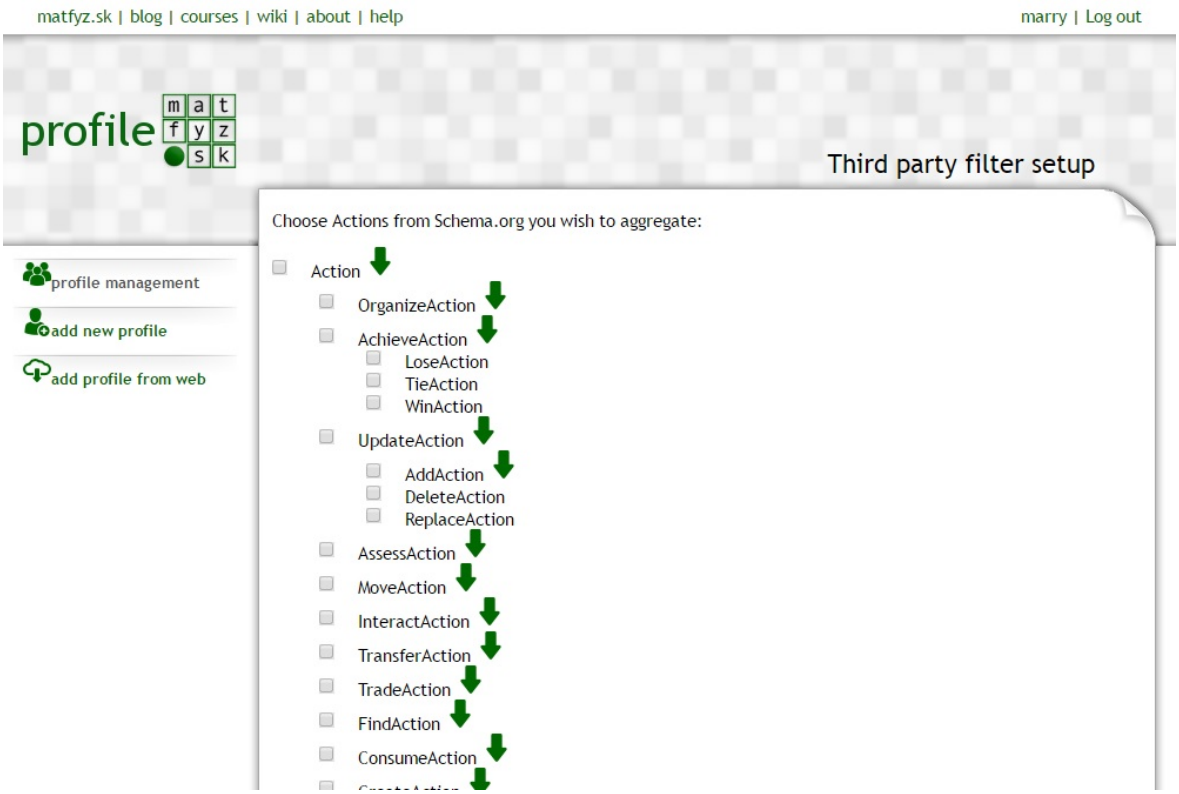

Figure 4.8: The set up of a filter for the third-party.

new attributes can be matched to the previously unmatched claims (this form is very similar to the one in the Choose profile prompt controller).

After the change of the conditions from the user side, the controller notifies the RP of these changes to the policy by sending an ID Token to their Update endpoint (or sends an ID token with the `ace` property set to "DELETED" if the user decided to revoke the access for this RP).

4.2.2.10 Third party management controller

When the user wishes to add a new third-party to a certain profile, the Profile management controller sends the user to the Third party controller's `add_new` function. The user is here prompted to enter the URL of the third-party, which is then verified for the `rel=''me''` attribute. Upon successful verification is the new third-party source stored into the database with the help of the `third_party_management_module`.

As already mentioned in the Profile controller, our implementation supports the usage of the Schema.org Action type. For each third-party the user creates a filter — a list of types (subtypes of *http://schema.org/Action*) they wish to aggregate from the third-party source. The form where the user picks the Actions they wish to have in the filter is depicted in the Fig. 4.8.

In this chapter, we have presented the most important components of our implementation. Other controllers, models and views can be found in the source code — see Appendix A.

Conclusion

In our thesis we dealt with the topic of the digital user identity on the Web, which consists of all the user accounts they keep across the web services. In particular, we focused on the two aspects: the management of all the identities of one user in one place and the communication between web-services where the user acts under these identities.

First, we have put together the requirements we place on effective usage of identities on the Web in general. We have defined seven main problems users face when having to create a new account for each service they want to use. We have then analysed the available systems and protocols, which address some of these issues or are otherwise related to the user identities. However, our analysis showed that these solutions solve only some of the critical problems. We have decided to take one of the existing internet protocols for user authentication – OpenID Connect and designed an extension for it, which supports the previously unconsidered issues. Our extension focuses on the exchange of user-related data among involved web-services. For example, when the user changes their profile picture at one service they use, with the help of our protocol extension, they can define that they want this change to automatically propagate to all the other cooperating services they use. Thus the user is freed from the tedious process of manual updating. Our design supports many other scenarios where communication among services is needed.

The other half of our thesis is devoted to the identity management system. We have designed and implemented a system for administration of the user identities in one place, which serves as an identity provider for other web-services of the user. Our provider lets the user manage all their identities in one place and yet keeps them completely separated to the outside world (e.g. professional and private identity). The user has fully under their control which information is revealed to the particular service and which stays hidden. In combination with the designed protocol, these two create a full package for management of digital user identity on the Web.

All the designs in this thesis have been successfully implemented. The proof-of-concept implementation with a demo web-service are available at `devprofile.matfyz.sk` and `demoapp1.matfyz.sk`. The source code is also available on the attached CD.

The proposed designs might be further extended to support other use-cases. For example, the concept of the identity management system might be used for e-government, where institutions need to access user data. The user would have all their information in one place, where they can effectively manage them. The doctor would have access only to health-related data, whereas the tax office to the earnings-related. Moreover, if the user changes their home address, this change would propagate to all the institutions having access to this information. Thus, the options of how to further build on our design are open and many useful applications might emerge.

Appendix A

Proof-of-concept Implementation

The source code of the proof-of-concept implementation, together with a demo web application is available at:

`http://gitlab.matfyz.sk/marry/master-thesis`

or on the attached CD.

The attachment includes:

devprofile.matfyz.sk *devprofile.matfyz.sk/*
: Implementation of the design proposed in Chapter 3. Includes external libraries and the CodeIgniter framework files. Our source files are located in *application/-controllers*, *application/models*, *application/views*, *js/* and *css/*.

demoapp1.matfyz.sk *demoapp1.matfyz.sk/*
: Implementation of a demo web-service which showcase the functionality of our design from the relying party's point of view.

Bibliography

[1] Tim Berners-Lee, James Hendler, Ora Lassila: *The semantic web*, in: *Scientific American* 284.5 (2001), pp. 28–37.

[2] John Bradley, Nat Sakimura, Michael Jones: *JSON Web Token (JWT)*, RFC 7519, Internet Engineering Task Force, 2015-05, URL: http://tools.ietf.org/html/rfc7519.html.

[3] Dan Brickley, Ramanathan V. Guha: *RDF Schema 1.1*, in: (2014), URL: http://www.w3.org/TR/2014/PER-rdf-schema-20140109/.

[4] Indie Web Camp: *rel-me*, URL: https://github.com/indieweb/rel-me.

[5] Michael Dowling: *Guzzle*, 2014, URL: http://guzzlephp.org (visited on 2016-03-12).

[6] EllisLab: *CodeIgniter*, 2015, URL: http://ellislab.com/codeigniter (visited on 2015-10-11).

[7] Facebook: *Graph API*, 2016, URL: https://developers.facebook.com/docs/facebook-login (visited on 2016-04-01).

[8] Anthony Ferrara: *password_compat*, URL: https://github.com/ircmaxell/password_compat.

[9] John Franks, Phillip Hallam-Baker, Jeffrey Hostetler, Scott Lawrence, Paul Leach, Ari Luotonen, Lawrence Stewart: *HTTP Authentication: Basic and Digest Access Authentication*, RFC 2617, Network Working Group, 1999-06, URL: http://tools.ietf.org/html/rfc2617.

[10] GitHub: *GitHub API: Oauth*, 2016, URL: https://developer.github.com/v3/oauth/ (visited on 2016-04-01).

[11] Google: *Google Identity Platform: OpenID Connect*, 2016, URL: https://developers.google.com/identity/protocols/OpenIDConnect (visited on 2016-04-01).

[12] Dick Hardt: *The OAuth 2.0 authorization framework*, RFC 6749, Internet Engineering Task Force, 2012-10, URL: http://tools.ietf.org/html/rfc6749.html.

[13] Ivan Herman, Ben Adida, Manu Sporny, Mark Birbeck: *RDFa 1.1 Primer - Third edition: Rich Structured Data Markup for Web Documents*, W3C Working Group Note, 2015, URL: http://www.w3.org/TR/2015/NOTE-rdfa-primer-20150317/.

[14] Ian Hickson: *HTML microdata*, W3C Working Group Note, 2013, URL: http://www.w3.org/TR/2013/NOTE-microdata-20131029/.

[15] Nicholas Humfrey: *EasyRdf*, URL: http://www.easyrdf.org/ (visited on 2016-03-12).

[16] IETF OAuth WG: *OAuth – OAUTH WG*, URL: https://www.ietf.org/mailman/listinfo/oauth (visited on 2015-03-27).

[17] Internet Assigned Numbers Authority: *JSON Web Token (JWT)*, 2016, URL: http://www.iana.org/assignments/jwt/jwt.xhtml (visited on 2016-04-03).

[18] ISO: *Information technology – Security techniques – A framework for identity management – Part 1: Terminology and concepts*, ISO ISO/IEC 24760-1:2011(en), International Organization for Standardization, 2011.

[19] Evan Prodromou James M Snell: *Activity Streams 2.0*, W3C Working Draft, World Wide Web Consortium, 2015-12, URL: https://www.w3.org/TR/activitystreams-core/.

[20] Roman Janajev: *Customizable and Usable Layout of blog.matfyz.sk Portal*, Master Thesis, Comenius University in Bratislava, 2014.

[21] Zhiwei Li, Warren He, Devdatta Akhawe, Dawn Song: *The Emperor's New Password Manager: Security Analysis of Web-based Password Managers*, in: *23rd USENIX Security Symposium (USENIX Security 14)*, San Diego, CA: USENIX Association, 2014-08, pp. 465–479, ISBN: 978-1-931971-15-7, URL: \url{https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/li_zhiwei}.

[22] Jeff Lindsay: *Web hooks to revolutionize the web*, 2007, URL: http://progrium.com/blog/2007/05/03/web-hooks-to-revolutionize-the-web/ (visited on 2015-10-14).

[23] LinkedIn: *Sign In with LinkedIn*, 2016, URL: https://developer.linkedin.com/docs/signin-with-linkedin (visited on 2016-04-01).

[24] Frank Manola, Eric Miller, Brian McBride: *RDF primer*, W3C Recommendation, 2004, URL: http://www.w3.org/TR/2004/REC-rdf-primer-20040210/.

[25] Matfyz.sk team: *About*, 2007-11, URL: http://www.matfyz.sk (visited on 2016-02-20).

[26] Nat Sakimura: *Dummy's guide for the Difference between OAuth Authentication and OpenID*, 2016, URL: http://nat.sakimura.org/2011/05/15/dummys-guide-for-the-difference-between-oauth-authentication-and-openid/ (visited on 2016-02-17).

[27] OAuth: *OAuth:Introduction*, 2007-09, URL: http://oauth.net/about.

[28] Luís Otávio Cobucci Oblonczyk: *jwt*, URL: https://github.com/lcobucci/jwt.

[29] OpenID Foundation: *Risk and Incident Sharing and Coordination WG homepage*, 2016, URL: http://openid.net/wg/risc/ (visited on 2016-04-06).

[30] Mária Šormanová: *Userprofile Module For matfyz.sk Portal*, Bachelor Thesis, Comenius University in Bratislava, 2013.

[31] Dave Raggett, Arnaud Le Hors, Ian Jacobs: *HTML 4.01 Specification*, W3C recommendation, 1999, URL: http://www.w3.org/TR/1999/REC-html401-19991224.

[32] N Sakimura, J Bradley, M Jones: *OpenID connect dynamic client registration 1.0*, OpenID Foundation Specification, 2014, URL: https://openid.net/specs/openid-connect-registration-1_0.html.

[33] N Sakimura, J Bradley, M Jones, E Jay: *OpenID Connect Discovery 1.0*, in: (2014), URL: https://openid.net/specs/openid-connect-discovery-1_0.html.

[34] Natsuhiko Sakimura, J Bradley, M Jones, B de Medeiros, C Mortimore: *OpenID Connect core 1.0*, OpenID Foundation Specification, 2014, URL: http://openid.net/specs/openid-connect-core-1_0.html.

[35] Schema.org: *Full Hierarchy*, 2011, URL: http://schema.org/docs/full.html (visited on 2015-01-20).

[36] Schema.org: *Getting started with schema.org using Microdata*, 2011, URL: http://schema.org/docs/gs.html (visited on 2015-01-20).

[37] Schema.org: *Welcome to Schema.org*, 2015, URL: http://schema.org/ (visited on 2015-01-20).

[38] Guus Schreiber, Yves Raimond: *RDF 1.1 Primer*, W3C Working Group Note, 2014, URL: http://www.w3.org/TR/2014/NOTE-rdf11-primer-20140225/.

[39] Brent Shaffer: *OAuth 2.0 Server PHP*, URL: http://bshaffer.github.io/oauth2-server-php-docs/ (visited on 2016-03-12).

[40] W3Schools: *W3Schools Online Web Tutorials*, 2016, URL: http://www.w3schools.com/ (visited on 2016-02-17).

[41] Web Data Commons project: *Web Data Commons - RDFa, Microdata, and Microformats Data Sets - December 2014*, 2014, URL: http://www.webdatacommons.org/structureddata/2014-12/stats/stats.html (visited on 2016-04-03).

[42] World Wide Web Consortium: *Social Web Working Group home page*, 2016, URL: `https://www.w3.org/Social/WG` (visited on 2016-04-01).