



Mathis Hesse, BSc

**Secure Authentication
based on Qualified Identities**

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Computer Science

submitted to

Graz University of Technology

Supervisor

Brenner, Eugen, Ao.Univ.-Prof. Dipl.-Ing. Dr.techn.

Institute for Technical Informatics

Faculty of Electrical and Information Engineering

Graz, October 2017

AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

Date

Signature

Abstract

Nowadays a high number of security-critical processes are performed online. To execute these kind of processes it is necessary to assure a person's identity through an authentication mechanism. For physical authentication a person can provide a photographic identification. Over the network it is not feasible to perform a full validation process of such an identification every time a security-critical resource is accessed.

As an alternative to assure one's identity online, it is necessary to provide an automated and secure authentication mechanism. In this thesis the development process of such a mechanism in form of a prototype is documented. To achieve this goal, relevant protocols and frameworks have been analyzed and compared to each other. Based on the insights from this analysis, a prototype has been designed and architectural details as well as the implementation decisions are explained. The secure authentication which this prototype provides is based on qualified electronic identities, which offer the same legal attributes as physical identification documents.

The result is a fully functional prototype based on OAuth 2.0, that offers secure authentication for users and is easily integrable for service providers.

Keywords:

Secure Authentication, Qualified Identities, OAuth 2.0

Acknowledgement

Diese Diplomarbeit wurde im Jahr 2017 am Institut für Technische Informatik an der Technischen Universität Graz durchgeführt.

An dieser Stelle möchte ich mich gerne bei meinem Betreuer Ao.Univ.-Prof. Dipl.-Ing. Dr.techn. Eugen Brenner bedanken, der mich während des Entstehungsprozesses der Masterarbeit begleitet und aktiv unterstützt hat.

Ein großes Dankeschön geht außerdem an die XiTrust GmbH. Hier gebührt mein Dank dem gesamten, großartigen Team, auf dessen Rat ich mich stets verlassen konnte. Ganz besonders möchte ich an dieser Stelle DI Gerhard Fließ hervorheben, der viel Zeit und Ressourcen aufgewandt hat, um mich bestmöglich bei allen anfallenden Themen und Fragen zu betreuen. Vielen Dank!

Ein riesiges Dankeschön richte ich hiermit auch an meine Familie und Freunde, die für die notwendige Unterstützung und auch Ablenkung gesorgt haben. Ganz besonders möchte ich Christina danken, die mir während dem Schreiben der Arbeit und vor allem im Endspurt eine groß Stütze und Hilfe gewesen ist.

Allen voran aber danke ich meinen Eltern, Werner und Gerhild, ohne die die Absolvierung meines Studiums in dieser Form nicht möglich gewesen wäre.

Vielen, vielen Dank!

Graz, Oktober 2017

Mathis Hesse

Contents

1	Introduction	1
1.1	Introduction	1
1.2	Motivation	1
1.3	Goals	2
1.4	Outline	3
2	Overview And Current Situation	4
2.1	XiTrust	4
2.2	Qualified identities and eIDAS	5
2.3	Authentication vs. Authorization	5
2.4	Multi Factor Authentication	6
2.5	Relevant Protocols	6
2.5.1	OAuth in General	7
2.5.2	OAuth 2.0	8
2.5.3	OAuth 1.0	11
2.5.4	SAML	13
2.5.5	OpenID	13
2.5.6	OpenID Connect	14
2.5.7	MOA-ID	15
3	Related Work	16
3.1	Existing implementations	16
3.1.1	Facebook	16
3.1.2	Twitter	19
3.1.3	Google	20
3.2	Threat Model	21
3.2.1	Client	21
3.2.2	Authorization- and Resource-Server	21
4	Details Of Research	23
4.1	Components of the prototype	23
4.2	Use Cases	23
4.3	Framework decisions	24
4.3.1	Token	24
4.3.2	Token Store	28
4.3.3	User ID Obfuscation	30

5	Experimental Evaluation	32
5.1	User Authentication	32
5.1.1	Obtaining the SAML-Artifact	32
5.1.2	Getting the SAML Assertion via the SOAP Interface	37
5.1.3	Processing the SAML-Assertion	38
5.2	OAuth 2 Flow: Communication between Client and API	39
5.2.1	Spring Client	40
5.2.2	PHP Client	42
5.2.3	Postman	44
6	Conclusion And Future Work	46
6.1	Conclusion	46
6.2	Future Work	46
6.2.1	Extend to OpenID Connect	47
6.2.2	Extend the API for additional tasks	47
6.2.3	Legal audit	47
6.2.4	Security audit	48
6.2.5	Developer Portal	48
	Nomenclature	50
	Bibliography	52

List of Figures

1.1	Identity- and Access Management with components in relation	2
2.1	Google Authenticator provides one-time tokens	7
2.2	Abstract flow of an OAuth Authentication	9
2.3	Detailed Authorization Code flow	10
3.1	Continue with Facebook Button	16
3.2	Process of activating external login for already existing account	18
3.3	Process of merging two already existing accounts	19
3.4	Sign in with Twitter Button	20
3.5	Sign in with Google Button	20
4.1	Components of the Prototype	24
4.2	Passing the user ID to the different clients without obfuscation	30
4.3	Passing the user ID to the different clients with obfuscation	31
5.1	Login	33
5.2	StartAuthentication interface between components	34
5.3	Standard template of the first step in MOA authentication step	34
5.4	Login prompt for Handy-Signatur	35
5.5	TAN prompt for Handy-Signatur	36
5.6	GetAuthenticationData between components	37
5.7	SAML response to redirect URL	38
5.8	Endpoints used by clients	39
5.9	Minimal UI of the PHP client	43
5.10	User data is returned and blank UI fields are filled	43
5.11	Getting new access token via Postman	44
6.1	Extending the prototype for additional tasks	47
6.2	Developer portal in context	48

Chapter 1

Introduction

1.1 Introduction

In modern times people don't only have one identity, they have many. Online there are different kinds of identities that are used for different services. First of all the term 'identity' needs to be defined. There are many different definitions of the word. In the introduction of the book *Electronic Identity* the author refers to articles, where the term is defined in 14 different ways. For their purpose, which matches the purpose of this thesis, the author decides to define identity as 'sameness' [dACA⁺14][p.3]. That means, that an individual can be identified at one point and recognized as the same person at a later point in time. Therefore in the context of this thesis users should be identifiable when logging into a portal. They should also be recognized again, when authenticating themselves at a later point in time.

As seen in figure 1.1, there is the term 'Identity and Access Management' (IAM). It consists of access management and identity management. This thesis will focus mainly on the topic of authentication, which is a subcategory of access management.

1.2 Motivation

As stated in the introduction, there exist different kinds of identities. The motivation is, to provide the users one, which is based on a qualified identity. This identity can be used to perform security-sensible actions online and can replace multiple identities within different portals. The currently existing alternatives are not sufficient, because they are often bound to certain registers. Therefore it is hard to acquire them, because often they are managed by government institutions and must fulfill strict requirements.

The Austrian Handy-Signatur as an example is bound to a specific person via the 'ZMR number', which correlates to an entry in the central register (also known as 'Zentrales Melderegister'). This means, that you have to be residing (or resided within the last 30 years) in Austria. Another way of using this service as a person not living in Austria, is to register at the 'Supplementary Register for Natural Persons' (ERnP) [Cen].

On the one hand this is an advantage, because the data is mostly up-to-date, but on the other hand users have to be registered in the ZMR or ERnP, when they want to use the Handy-Signatur service. Also when it comes to interactions between different countries,

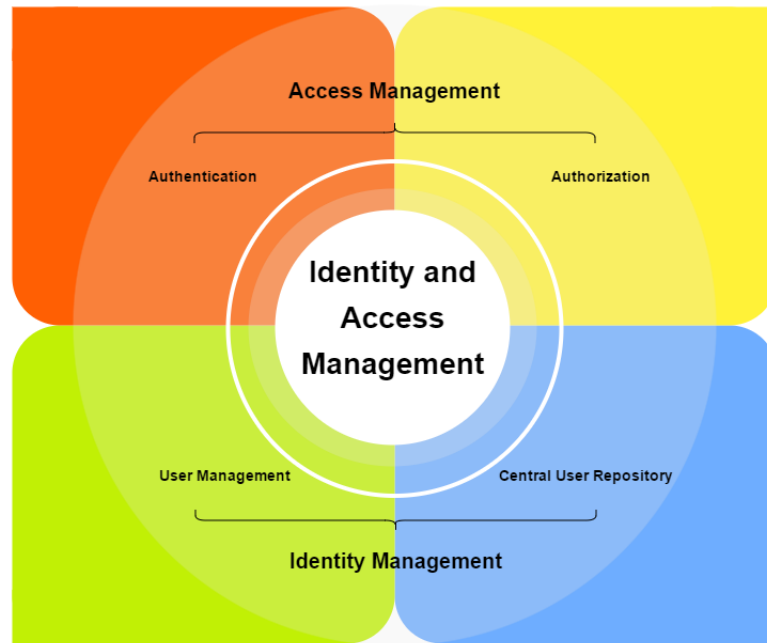


Figure 1.1: Identity- and Access Management with components in relation

the acceptability of electronic identities is a controversial topic.

The motivation is, to offer users an easy way to login with their qualified identity without the restrictions of being citizen of a certain state or participate in a registration process which would take up a lot of time.

1.3 Goals

The Goal of this thesis is, to implement a working prototype for providing an application programming interface (API) for a secure login mechanism to companies and their users. This secure login mechanism should be based on qualified identities. The challenge is, to fulfill all necessary security prerequisites and still keep the system simple to integrate and use. To achieve this, the thesis was carried out together with an identity provider, which can provide the qualified identities our system is based on. In this case these identities are provided by the company XiTrust through their product 'XiIdentity', which they created in cooperation with the company A-Trust.

The outcome is a working and functional prototype. It demonstrates, that users are able to use the implemented service to login to portal with their qualified identities. Therefore also sample clients are implemented to demonstrate the communication with the prototype.

1.4 Outline

First of all an overview of the current situation is provided. In this overview important terms are defined and relevant topics are discussed. Several protocols and frameworks are introduced, which will be considered to be part of the implementation of the practical part of this thesis.

The advantages and disadvantages of the highlighted methods will influence the decisions which are necessary, when planning the architecture of the prototype. The details of this architecture are explained in chapter 4. Here the decisions for the implementation phase are explained and the blueprints for the prototype are made.

In chapter 5 the actual progress of the implementation is transcribed. The interaction of the outlined components will be implemented and documented. Interfaces are explained and the flow of an authentication process is divided in smaller modules to provide a clean overview of the prototype.

As there already exist similar implementations, they have been investigated in chapter 'Related Work'. Former mistakes which occurred in the handling of said implementations are evaluated and solutions are extracted. Additionally different threat models are analyzed and taken into account.

In the last section the findings of our research, as well as the discoveries of the implementation process are concluded. Certain steps are outlined to improve the implementation and convert it into a product. Finally the directions in which further research could go, are listed in this chapter.

Chapter 2

Overview And Current Situation

In this chapter the current situation in the field of Identity Management and authentication/authorization mechanisms is discussed.

Furthermore the company XiTrust is introduced. An overview over the existing products with the main focus on XiDentity is given, which is important for the scope of this thesis.

2.1 XiTrust

The company XiTrust exists since 2002 as successor of the former Xicrypt GmbH. XiTrust offers their customers the possibility, to apply a framework for streamlining electronic processes.

The main idea is, that a lot of companies rely on a form of document processing, that is not up-to-date, because there is a media disruption in the process. Meaning, that when signing or archiving documents, the companies first print the documents, then sign them, only to have them digitalized afterwards. This is not necessary in modern times and XiTrust offers the following products, to optimize these processes [Gmb]:

XiTrust Moxis is a product for signing and maintaining digital documents. The signature is legally equivalent to a handwritten signature. Moxis is a web-based system which is integrated in the IT infrastructure, so no document has to leave the customers ecosystem.

XiTrust Business Server is centrally entered into the customers IT infrastructures to offer a broad range of services. With the XiTrust Business Server (XBS) companies can sign, check and archive their documents. Furthermore secure email communication can be applied by integrating encryption and digital signatures into the email-infrastructure. It is also possible to create workflows for automating business procedures and integrate E-Government within the processes of the company.

XiTrust Time Stamp Server is a service, that offers the generation of electronic time stamps / time signatures in the data center of the customer. Therefore time stamps can be requested and produced to secure documents and files.

XiDentity is a product with which it is possible, to obtain a qualified, digital identity. This digital identity is eIDAS-compliant (for details see next section). For this process, the user has to sign up at XiDentity. A so called registration officer (RO) has to confirm the identity of the registrant. This process is done in person, where the registered RO issues the user a certificate after crosschecking the user's data from a photographic identification document. This certificate represents your qualified digital identity.

It has recently become possible to complete the registration procedure online. For this scenario the registration officer will induce a video-chat with the users and check their identity via specified methods remotely.

2.2 Qualified identities and eIDAS

eIDAS (electronic IDentification, Authentication and trust Services) is an EU regulation which contains a set of standards concerning electronic identification and trust services for electronic transactions in the internal market. This regulation with the number 910/2014 from 2014 ensures, that trust services like electronic signatures and website authentication mechanisms work across borders of EU countries and have the same legal status as their paper based equivalents. Most of the contained provisions came into effect in July 2016 [PtCotEUb].

Users, who obtained such an electronic identity (in our case through the process described in the previous section) should be able to make use of it in several different places on the internet. For this to work, operators of service portals need an easy way of integrating an authentication mechanism to their sites. Right now it is rather hard to integrate such a mechanism into a portal, but with the aid of the prototype developed within the scope of this thesis, the effort and costs of the integration are minimized.

2.3 Authentication vs. Authorization

It is important for further understanding to distinguish authentication and authorization. These terms are often mixed up in the field of identity access management (IAM) but it is crucial to know the differences and synergies when implementing an API for an identity provider (IdP), to separate the responsibilities between the involved parties [BKE14].

Authentication is the process of ascertaining if a user really is who he or she claims to be. So for an Identity Provider the task is, to authenticate and identify the users rather than authorizing rights to them. This goal can be achieved in many different ways with various authentication mechanisms and protocols, which will be covered in the following sections.

Authorization on the other hand, goes one step further. Users can be granted certain rights to perform actions or access resources depending on their role within a system. That means, that as a first step of authorization, the user usually is already authenticated. Sometimes authorization mechanisms are utilized to perform authentication, as often seen with OAuth 2.0 (for details see section 2.5.2).

2.4 Multi Factor Authentication

Another important term in this context is multi-factor authentication. Today's authentication mechanisms are based on following factors [dB12][BKE14]:

- Something you know
- Something you have
- Something you are
- Something you do

Basic authentication by username and password is based on the factor 'Something you know'. For sensible accounts and operations this often does not suffice, therefore additional factors have to be added to the authentication process. When using two of these factors, it is called 'Two Factor Authentication' (2FA).

The factors 'Something you are' and 'Something you do' are getting more and more popular. Examples for the first mentioned factor would be biometric attributes like fingerprints or iris patterns. The factor 'Something you do' would recognize one's voice, handwriting or typing patterns. These factors are somehow problematic, as it is hard to revoke or reissue templates that rely on a person's attributes [JNN08].

The 'Something you have' factor is the most common additional factor when it comes to 2FA. Special hardware-tokens can be provided, that generate one-time keys that can be used as additional authentication mechanism. Often these hardware tokens are outsourced to software emulated hardware tokens e.g. as applications on smartphones. An example for such an application is the 'Authenticator' from Google. One-time tokens are continuously generated for each added application, as seen in figure 2.1.

This is an easy way to distribute tokens, as a mobile phone is something most users nowadays own. Therefore it is also common to verify an authentication process via a person's mobile phone number e.g. as short message or automated call. When using the phone as additional factor for authentication, the user should be aware, that once the device is compromised and used for the login process, it could lever out the 2FA. This happens when using the same device for entering the password and receiving the token for the additional authentication step.

2.5 Relevant Protocols

As Phil Zimmermann, creator of PGP stated:

'Anyone who thinks they have devised an unbreakable encryption scheme either is an incredibly rare genius or is naive and inexperienced.' [Zim91]

This principle not only holds for encryption schemes, but can also be applied to security implementations in general. So as usual in the field of IT security the solution is based on standardized protocols instead of implementing a proprietary solution. There exist various established and widely accepted protocols for authentication and authorization, which will be discussed below.

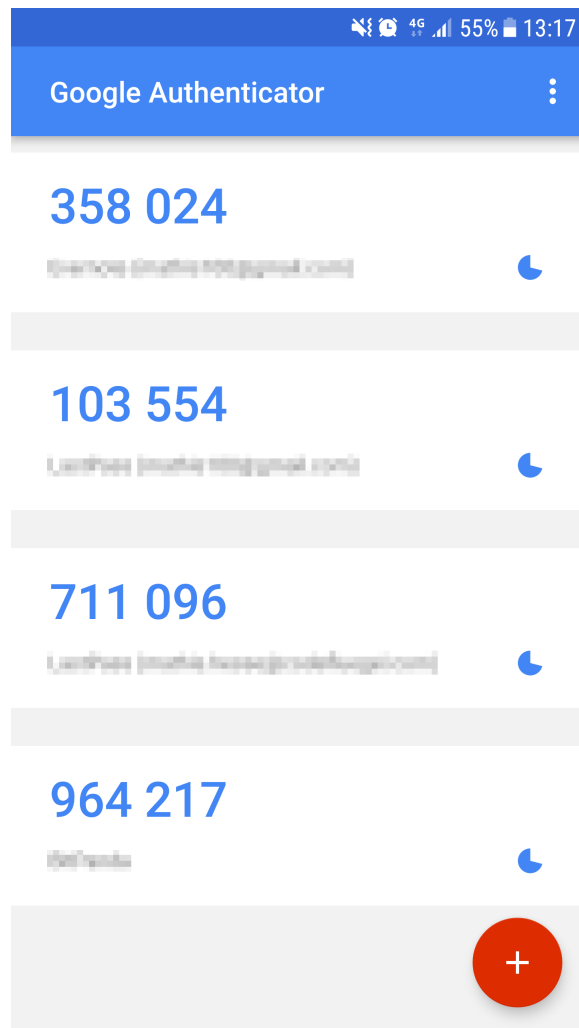


Figure 2.1: Google Authenticator provides one-time tokens

2.5.1 OAuth in General

There exist two versions of OAuth: Version 1.0 and Version 2.0. As the version number suggests, 2.0 is the successor of 1.0, but they are completely different in means of how they are working. This is also the reason that they are not compatible. Also OAuth 2.0 hasn't fully replaced OAuth 1.0, as some companies still rely on it (e.g. Twitter) [Twi].

The main improvements between version 1.0 and 2.0 are, that OAuth 2.0 offers more flows, which allows better support for non-browser based applications, e.g. smartphone apps. It is also no longer required to support cryptography as a service provider, as the security of OAuth 2.0 relies on a secure connection based on SSL/TLS. There exist specifications for OAuth 1.0 [HL10] as well as OAuth 2.0 [Har12][JH12], which were used to gather the following information.

2.5.1.1 Roles

In the terminology regarding OAuth, there are so called 'two legged' and 'three legged' implementations. By having a look at the 'two legged' implementations, only the server and the client are involved in the flow, which is not sufficient for our field of application. Whereas our focus will be on the 'three legged' implementations, where also the resource owner is part of the flow. The roles used in the process are defined as followed:

Resource Owner is usually the user, who grants an application access to the account they own (therefore the term 'Resource Owner'). This access can be limited to the scope of the granted rights (e.g. read, write, ...).

Authorization Server verifies the users identity and handles the authentication tokens. The authorization server is also responsible for maintaining clients. So for requesting data from the user, the application has to be registered. This means that the authorization server has to create and manage the API keys and secrets for several clients, which are necessary for the authentication of the client itself. Also the redirect URLs for each client are stored here.

Resource Server holds protected resources, which can be accessed via an access token. Often the Authorization- and the Resource-Server are combined in one server instance.

Client is the application, that wants to authenticate the Resource Owners and access their resources.

Details of the two different versions are introduced in the following sections. At first OAuth 2.0 is explained in detail, as it is more relevant for our field of application.

2.5.2 OAuth 2.0

As OAuth 2.0 is an important protocol when it comes to the use-case that are applicable in this context, the details of it are highlighted in the following section.

The OAuth 2.0 authorization framework enables a third-party application to obtain limited access to an HTTP service, either on behalf of a resource owner by orchestrating an approval interaction between the resource owner and the HTTP service, or by allowing the third-party application to obtain access on its own behalf. This specification replaces and obsoletes the OAuth 1.0 protocol described in RFC 5849. [Har12]

2.5.2.1 OAuth 2.0 Flow

To understand the OAuth authentication itself, an abstract flow of an authentication process is displayed. In Figure 2.2 the interaction between the introduced roles is shown.

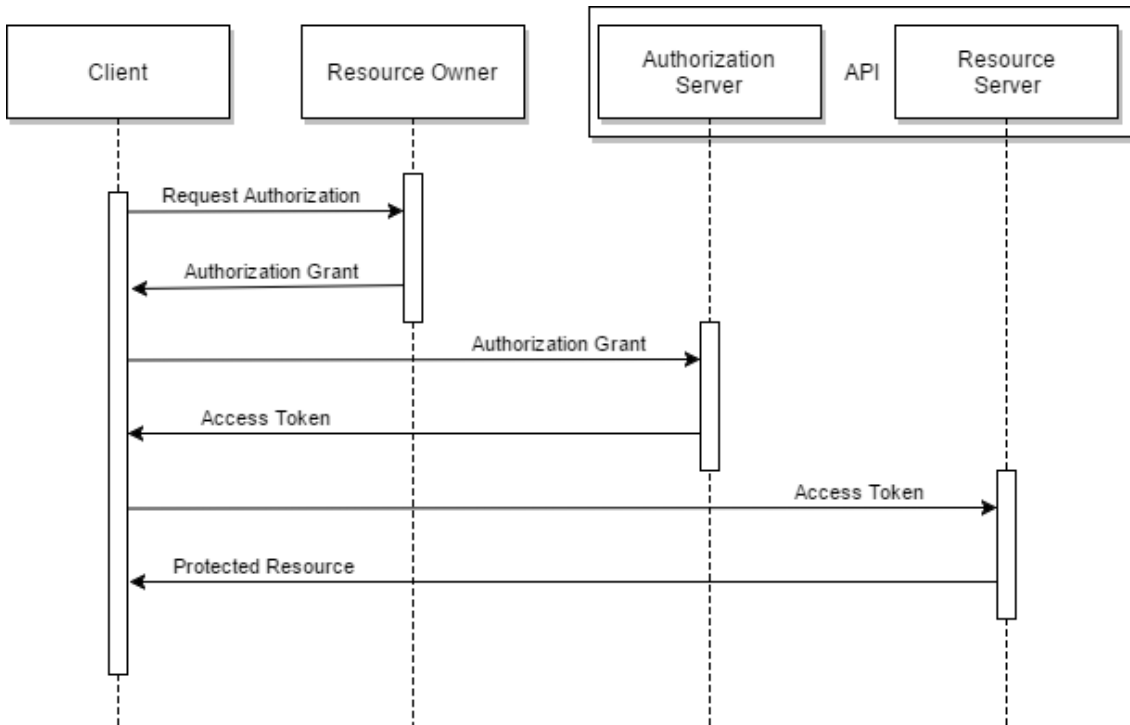


Figure 2.2: Abstract flow of an OAuth Authentication

The following steps are necessary:

1. The Client application wants to access resources from the user and asks for permission.
2. The User can decide, if the application is trustworthy. If so, the user accepts the request and the application gets an authorization grant.
3. The application proceeds by sending the obtained authorization grant together with the identity of the application to the authorization server.
4. If the received data is correct and verified, the authorization server generates an access token and sends it back to the application. After this step, the authentication is completed.
5. For accessing a protected resource, the application sends a request, that includes the access token, to the resource server.
6. After the resource server validated the access token, the application is granted access to the resource.

2.5.2.2 Authorization Grant

As seen in Figure 2.2, the authorization grant is sent between the participating roles. As for now, the concept of an authorization grant was kept abstract, because there are

different types of authorization grants. Depending on the scenario, it is important to choose a reasonable grant type for the situation.

Authorization Code is the most common version of authorization grant. A condition for using this type is, that the application is able to keep the API secret save, which is the case when working with server-side applications. In Figure 2.3 the detailed flow of the authorization code grant type is visualized.

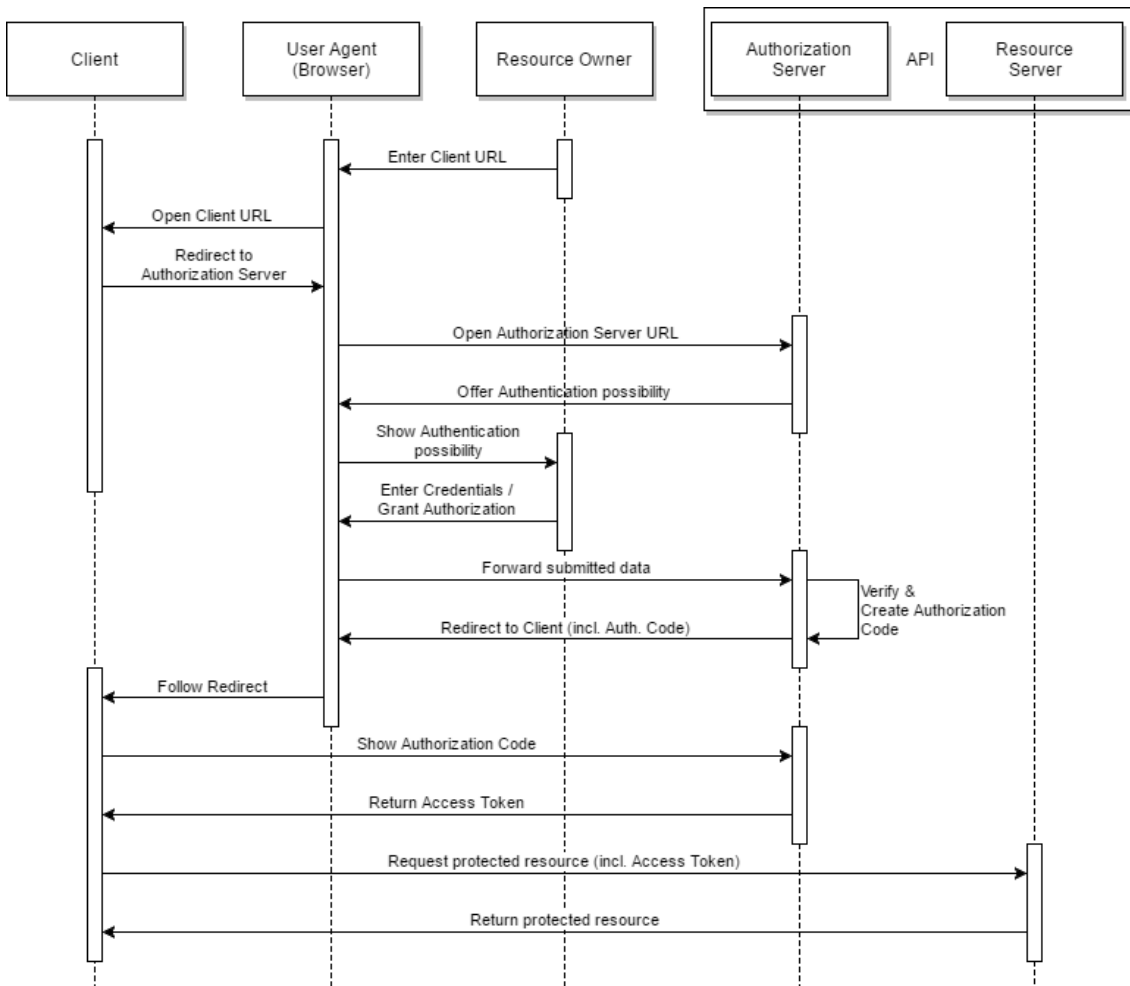


Figure 2.3: Detailed Authorization Code flow

- After visiting the application link through the browser, the user gets a link, which includes the endpoint of the authorization server, the API key, a callback URL for redirection and other parameters like the response type and the scope of the request.
- When the user clicks on this link, he or she is asked to login at the authorization server. After login, the user normally is presented with a prompt, in which he or she can decide if access to certain data should be granted to the application.

- If everything went successful, the user is now redirected to the specified redirect URL including an authorization code.
- The application can now send a request to obtain an access token to the authorization server. This request contains the API- key and secret, the authorization code and a callback URL.
- If the request is valid, the authorization server sends back a response containing an access token with other details (like an expiration date, and optionally a refresh token, which can be used to obtain a new access token when expired). Now the application can use this access token (until expired or revoked) to access protected resources.

Implicit as authorization grant type works similar to an authorization code grant. The main difference is, that the authorization server does not return an authorization code, which can be exchanged for an access token. Instead it returns the access token directly. This is particularly useful, because the application does not have to include the API secret in the request, which is interesting, when the application is e.g. an app on a mobile phone or a web-application, where the API secret can not be stored safely, because the user could gain access to it. Different from the authorization code grant, the implicit authorization grant does not support refresh tokens.

Resource Owner Credentials is rarely used, because it requires total trust in the application. The resource owners give away their private credentials to the application, which uses them to further get an access token. This access token can be utilized to obtain resources or functionalities that require authentication. The application has full control to the resource owners account and may exploit their power. Also if the user reuses the credentials on other sites, the application might also be able to login to these services if they have malicious intentions.

Client Credentials are used, when the application needs access to certain functionality or resources, that are not related to a resource owner. This is the case, if it is enough, that only the application needs to be authenticated. The authentication usually happens by presenting the authorization server the applications API- key and secret.

2.5.3 OAuth 1.0

OAuth 1.0 was stabilized at version 1.0 in October 2007. In June 2009 it was revised (Revision A) and thereby referenced as OAuth 1.0a (for simplicity reasons we will reference to the revised version as OAuth 1.0 throughout this thesis). As OAuth 2.0 is based on OAuth 1.0, there are a lot of similarities like the basic idea of the flow. One of the main differences is, that there is a signing mechanism in place for requests.

OAuth provides a method for clients to access server resources on behalf of a resource owner (such as a different client or an end- user). It also provides a process for end-users to authorize third- party access to their server resources

without sharing their credentials (typically, a username and password pair), using user- agent redirections. [HL10]

The roles correspond to the roles used in OAuth 2.0. Unlike in OAuth 2.0 the authorization- and resource server are not treated separately, but simply called 'server'. Also in prior versions of the specification another terminology was used, where the client was called 'consumer', the resource owner was stated as 'user' and the (resource) server was known as 'service provider'.

2.5.3.1 OAuth 1.0 Flow

The first step in OAuth 1.0 is to obtain a request token. For this a request to the `request_token` endpoint is made. This request looks different than the one in OAuth 2.0 because it includes completely different parameters:

oauth_consumer_key is the equivalent to the API key / `client_id` from OAuth 2.0.

oauth_timestamp is a unix timestamp, which has to be greater than the one in the last request.

oauth_nonce is a random string, which is generated for each request. In combination with the timestamp it helps to prevent replay attacks which would be possible when the requests are made over a non-secure channel.

oauth_signature is the signed request. For Details see section 2.5.3.2.

oauth_signature_method specifies the algorithm used, for creating the `oauth_signature`. There are three methods defined (HMAC-SHA1, RSA-SHA1, and PLAINTEXT) but the service provider can implement their own methods as well.

oauth_version defines the version of OAuth used. This parameter is optional, but if present, is has to be the value '1.0'.

oauth_callback is the redirect URL, where the resource owner is redirected by the server after the authorization step.

optional parameters exist, but are not relevant for the context of this thesis.

After successfully completing this step, a request token (`oauth_token`) and a `oauth_token_secret` are returned. The user is then redirected to the authorization URL using the retrieved `oauth_token`. After an successful authentication, the users grant access to the client and are then further redirected. Using the `oauth_token`, another signed request is sent to exchange the request token for an access token, which can be used to access protected resources on the server (in combination with the `oauth_token_secret` used to sign the requests).

2.5.3.2 Signing process

The signing process is the main difference between the two OAuth versions. It is simultaneously the reason, why OAuth 1.0 is still used in some cases and why it got replaced by OAuth 2.0.

The process helps, to keep the `consumer_secret` safe, even if operating over a non-secure channel like HTTP. But at the same time the implementation is error-prone. Every request must be signed by the client and verified by the server in order to prevent parties from making unauthorized requests. To sign the request, the client has to follow very specific steps. Every key value pair (excluding the `oauth_signature` that is created in this step) from the request, as listed in the prior section, has to be percent encoded and listed lexicographically by the key in a so called 'parameter string'. To get a full 'signature base string', the HTTP method used (GET or POST) and the URL have to be included beside the parameter string. They have to be conjoint by percent encoding all the three parameters and append it with a '&' character in between.

To obtain the key for signing this signature base string, the client has to join the `consumer_secret` and the `oauth_token_secret` (again by appending them via the '&' character) and use it to sign the string with the specified algorithm. If the `oauth_token_secret` does not exist yet, an empty string is appended to the consumer secret. After this, the `oauth_signature` is created and can be further used as BASE64 encoded string.

2.5.4 SAML

The first version of SAML with version number 1.0 was adopted as a standard in November 2002 by OASIS. A year later a newer version with minor changes was introduced as SAML 1.1. In March 2005 OASIS ratified SAML 2.0 as a standard. The newer version is a convergence of SAML 1.1, the Identity Federation Framework (ID-FF), which was donated to OASIS by the Liberty Alliance and Shibboleth 1.3. SAML 2.0 is incompatible with its predecessors. It is defined as follows:

The OASIS Security Assertion Markup Language (SAML) standard defines an XML-based framework for describing and exchanging security information between on-line business partners. This security information is expressed in the form of portable SAML assertions that applications working across security domain boundaries can trust. The OASIS SAML standard defines precise syntax and rules for requesting, creating, communicating, and using these SAML assertions [RHPM06].

SAML is mainly used for single sign on (SSO) inside enterprise infrastructures. It is not meant to be used for mobile- or native applications. In comparison to other protocols like OAuth it lacks in flexibility and is rather hard to implement. Some parts of SAML are still used in chapter 5, as it is part of MOA-ID (see section 2.5.7).

2.5.5 OpenID

OpenID is an open and decentralized protocol for authentication. The protocol is still rarely applied in existing websites. Therefore it is shortly explained for the sake of com-

pleteness, even if it is declared obsolete by the OpenID foundation, because of its successor OpenID Connect, which will be discussed in the next section [Foua].

For authentication, users have to create an account at a OpenID provider of their choice. After successful registration the users receive an authentication URL, also called 'OpenID identifier'. This identifier can be used at online applications (in the OpenID context they are called 'relying parties'), which support OpenID authentication. As an example, when the user 'exampleuser' registers at OpenID provider 'exampleopenidprovider', the identifier could be assembled the following way:

```
www.exampleuser.exampleopenidprovider.com
```

The user can now try to access a protected resource at the relying party and therefore enter the identifier, after being asked for a login. After entering the URL at the relying party, the user is required to authenticate at the OpenID provider. In the following step, the user is asked if he or she trusts the relying party and which information should be provided to them. The user is now redirected back to the relying party with the result of the authentication process, which can either be the information of the authenticated user or an error message.

2.5.6 OpenID Connect

OpenID Connect is the successor of OpenID. The protocol was completely renewed and is explained by the OpenID foundation as follows:

OpenID Connect 1.0 is a simple identity layer on top of the OAuth 2.0 protocol. It allows Clients to verify the identity of the End-User based on the authentication performed by an Authorization Server, as well as to obtain basic profile information about the End-User in an interoperable and REST-like manner.

OpenID Connect allows clients of all types, including Web-based, mobile, and JavaScript clients, to request and receive information about authenticated sessions and end-users. The specification suite is extensible, allowing participants to use optional features such as encryption of identity data, discovery of OpenID Providers, and session management, when it makes sense for them. [Foub]

As mentioned above, OpenID Connect is based on OAuth 2. Therefore the detailed flow will not be part of this section, as it is already discussed in section 2.5.2. In addition to OAuth 2, OpenID Connect provides an so called 'ID Token', besides the already known token types. This token is a JWT (for details see section 4.3.1.3) which contains information about the authenticated user. A disadvantage of OpenID Connect is, that because the specification is relatively young, it lacks of implementations and libraries for integrating it. It will take time for it to become widely used and to be integrated in different programming languages in the form of stable libraries.

2.5.7 MOA-ID

The Austrian E-Government is providing 'Modules for Online Applications' (MOA) to access and integrate their tools and services. MOA-ID is one of these modules and is used for authentication, as stated in the official description:

This module facilitates the secure and unique identification and authentication of users who process online procedures with a Citizen Card. The authentication is carried out by using the qualified signature as well as the identity link of the Citizen Card and therefore has the highest level of security. With it, a login with the Citizen Card is possible to areas that have sensitive data stored. Examples of this are inspecting files and accounts, banking and transaction systems, as well as other areas in which personal data is stored. [EGICEI]

This explanation only addresses authentication with citizen cards, but it is also possible to use MOA-ID in combination with XiDentity. This will be our main scenario for the prototypical implementation. While MOA-ID is not considered as interface between the prototype and the client, it is still an important framework as we use it to integrate the authentication process into our implementation. The exact flow is described in chapter 5.

Chapter 3

Related Work

In this chapter existing implementations for similar use cases like our prototype supports are discussed. Also different papers regarding OAuth 2 security are examined for a further understanding which security flaws may be exploited in the implementation of the prototype.

3.1 Existing implementations

As the idea of an external login mechanism has become popular among users and service providers, a lot of different implementations for this use case exist. Dominant in this area are the social network providers like Facebook, Twitter and Google, as many people use their services and already own an account on these sites. These accounts can then be used on many sites for authentication. Nevertheless, the provided identities in these cases are not qualified identities. It is highlighted, how the most popular providers of external login modules implement their services. As a next step they are analyzed and compared to each other. Also if the documentations provide useful information relevant to our prototype, they will be discussed at this point.

3.1.1 Facebook

Facebook is the most used social network worldwide. Statistics show, that in the second quarter of 2017 there are 2 billion monthly active users (users, that logged in to Facebook during the last 30 days) [Por]. Facebook offers their external login for service providers via 'Facebook Login', formerly known as 'Facebook Connect' [Fac]. Buttons, like the one shown in figure 3.1 show, that the external login via Facebook is available on a portal.

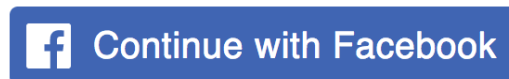


Figure 3.1: Continue with Facebook Button¹

Facebook mainly uses a proprietary SDK for their login mechanism. Their documentation of the external login does not mention OAuth 2 in the higher level articles. But when taking a closer look at it or after implementing their login, it is obvious that the implementation fulfills the OAuth 2 specification and is therefore also compatible with it.

Some valuable information can be found in the security- and advanced section of the documentation. It is stated, that the `application secret` should never be included in client-side code (e.g. JavaScript), or code that could be decompiled (e.g. native applications). This also holds for implementing OAuth 2 as a client. In these scenarios the implicit flow must be used.

Also when it comes to combining the Facebook Login with a conventional login (consisting of username and password), there are useful tips of merging the accounts of users using both variants. This is highly relevant to our case, as the prototype is integrated in existing environments, where another form of authentication could already be in place. There are different possible scenarios when it comes to merging accounts.

First of all the user could already possess an conventional account and wants to enable the external login. In this case, that is visualized in figure 3.2, the already authenticated user should be provided with an option to additionally login via the external method. This way the system has to merge the two accounts into one. Facebook suggests to create a new table with the Facebook specific data in the existing database, as it is easier to maintain and also to add other external login providers afterwards. After linking the two accounts it will be possible for the user to login either via conventional- or external-login in the future, as the system has mapped both variants to one account.

¹<https://developers.facebook.com/docs/facebook-login/best-practices> (accessed September 04, 2017)

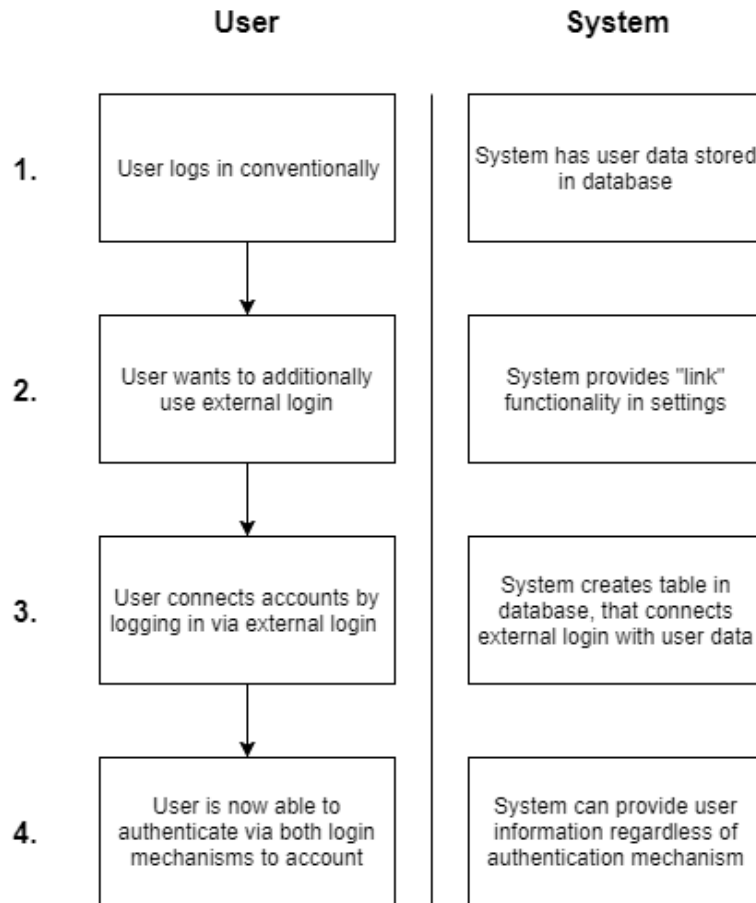


Figure 3.2: Process of activating external login for already existing account

Another case is, when the user has created an account via the conventional method as well as via the external method. In this case, two accounts exist, that aren't connected. This scenario, as visualized in figure 3.3, is complex and requires an explicit section for account merging (which data should be used from which account etc.). As this case is not so common (as the users usually know, that they already created an account on the site) not many service providers offer such a merging functionality but rather suggest to delete one account and then link the account again via the method discussed above.

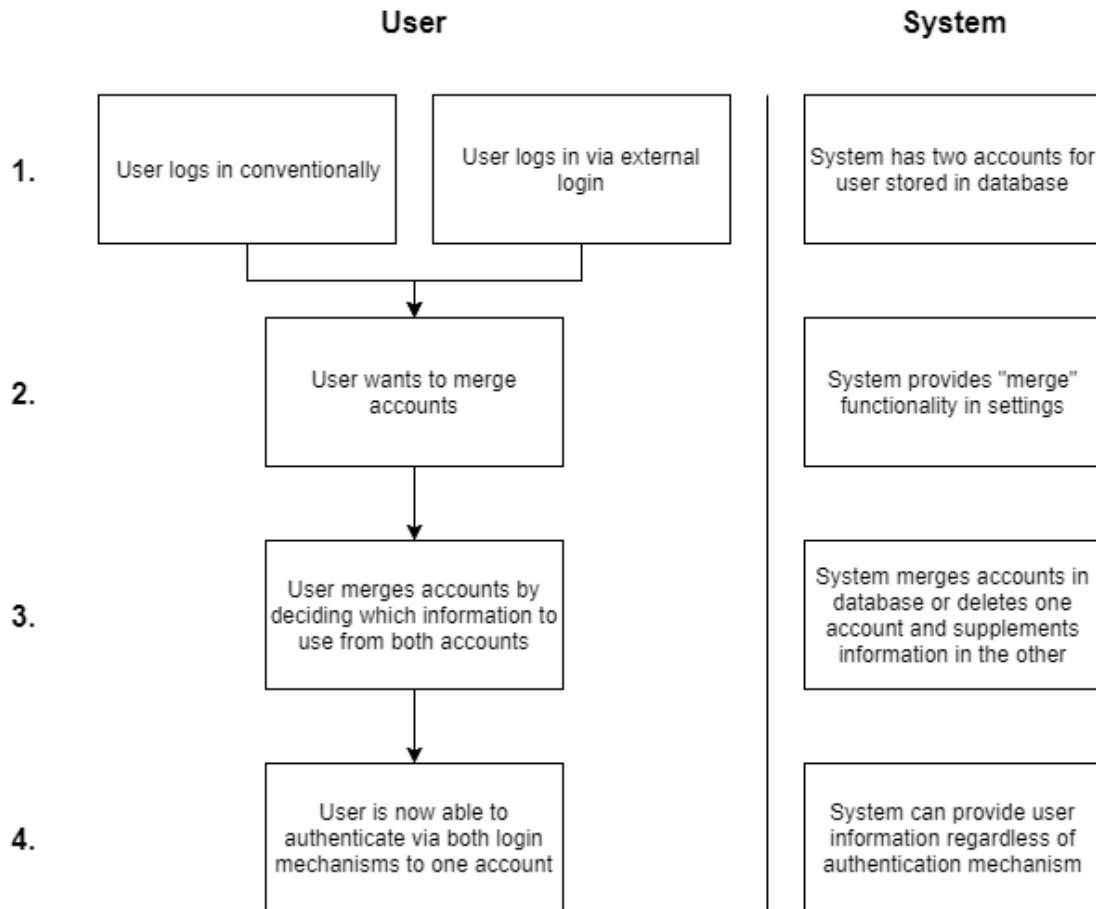


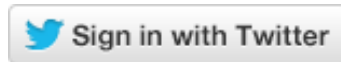
Figure 3.3: Process of merging two already existing accounts

Of course also the scenario exists, where the user initially logged in via the external authentication mechanism and then wants to add a conventional login. This case is similar to the first discussed scenario (see figure 3.2) and can be handled accordingly.

3.1.2 Twitter

Twitter is another large social network, that provides external authentication via their site. Usually service providers use the officially provided 'Sign in with Twitter'² (see figure 3.4) button to express, that the external login via twitter is available.

²<https://dev.twitter.com/web/sign-in> (accessed September 04, 2017)

Figure 3.4: Sign in with Twitter Button³

Twitter is one of the larger providers for external authentication, that still uses OAuth 1, as already mentioned in the previous section 2.5.1. The documentation is well structured and comprehensive. As OAuth 1 was considered, but ruled out as protocol for our implementation, details for the implementation itself are negligible. Still, when it comes to the documentation and visualization for the developers, Twitter could serve as role model.

3.1.3 Google

Google offers different ways to implement an external authentication with their accounts, therefore they provide the most flexible solution of the mentioned companies. If external login is the only module, that is required, developers can use the client library 'Google Sign-In'⁴. It is based on OpenID Connect and widely used on different websites (these implementations can be found by the distinctive button shown in figure 3.5). Google Sign-In is a part of the Google Identity Platform. There are guides to integrate this mechanism into native mobile applications as well as websites and other devices.

Figure 3.5: Sign in with Google Button⁵

Google also offers plain OAuth 2 to access their APIs. A exhaustive documentation is in place and there even is a 'OAuth 2.0 Playground'⁶, with which developers can dynamically test the interaction with Google's OAuth 2 interface. An important sentence in the documentation states, that:

One of the advantages of using OAuth 2.0 for authentication is that your application can get permission to use other Google APIs (such as YouTube, Google Drive, Calendar, or Contacts) at the same time as you authenticate the user.

The consideration, that the authentication mechanism comes along with the authorization process shows, that authorization protocols can be used for authentication. This way the resulting implementation is also very easy to extend.

³<https://dev.twitter.com/web/sign-in> (accessed September 04, 2017)

⁴<https://developers.google.com/identity/> (accessed September 06, 2017)

⁵<https://developers.google.com/identity/branding-guidelines> (accessed September 06, 2017)

⁶<https://developers.google.com/oauthplayground/> (accessed September 06, 2017)

On top of this OAuth 2 implementation a OpenID Connect layer is present, on which Google Sign-In is based. This layer is not only accessible through the mentioned client libraries, but also as standardized OpenID Connect endpoints.

The documentation also includes implementation guides and example libraries for various fields of applications and programming languages.

3.2 Threat Model

As OAuth 2 is a popular protocol, many researchers spent time and resources on analyzing the security aspects of it. A lot of the gathered information from related papers is presented in a document which discusses the threat model and security considerations regarding OAuth 2 [LMH13]. Also the threat model and the different attack vectors are discussed, while omitting obvious attacks like keyloggers, device theft, phishing and others.

3.2.1 Client

The most common threat model regarding clients concerns the client secret. It should always be kept secret, as it can be used by an attacker to bypass client authentication at the authorization server. This is not a security risk for the user, as the user authentication is not affected by the client secret, rather than for the client. Leaking of the client secret often happens, when the client implements the wrong grant for the application. When the implementation can keep the secret safe and the connection is secure, the secret should not leak. However if the client stores the secret in a decompilable application or a client-side implementation, the secret can be exposed. Also if the connection is insecure, the client secret can be intercepted during transmission. When the secret was leaked, the client should have the possibility to reissue the secret for the application.

Another scenario is, that the attacker has additionally overcome some of the applications security controls and obtains user related data, which may include stored access or refresh tokens of users. The attacker can use the access token directly, to access resources on the resource server or in case of authentication, to impersonate the user. In combination with the client secret the refresh tokens can be used to request a new access token and therefore the attacker is also able to execute above mentioned actions. To prevent this, it should be possible to revoke issued tokens.

3.2.2 Authorization- and Resource-Server

The authorization server should always require the client to enter a full redirect URL. It is possible to rely on the URL the server gets as parameter via requests, but hereby no validation is in place. This way the attacker could gain access to authorization codes or access tokens by manipulating the redirect parameter to a URL, which leads to a malicious server.

If the attackers have access to the authorization- or resource server, they are able to add new access tokens, access all the already issued tokens or just freely access the resources of the users. This way it would be easy to compromise the whole system. This is why it is very important to keep the servers safe in general. In the case of our prototype, there are no user resources stored on the resource server. The data of the users which

is used for authentication is volatile and only available as session information, when the user is authenticated. This way a compromised resource server would not have such severe consequences, like it would have in the situation, where the user data is stored permanently.

Chapter 4

Details Of Research

After reflecting upon the different protocols and frameworks in chapter 2, and learning about existing implementations in chapter 3, OAuth 2 was chosen as protocol for the prototype. The decision was mainly made between OAuth 2 and OpenID Connect. The reason, why OAuth 2 was chosen is, that at this point in time there exist more libraries for it and it has a broader community. OpenID Connect has some features which would make the authentication a little bit more convenient, but as we do not need to support multiple identity providers or have flexible identity attributes, a lot of OpenID Connect's advantages are negligible. Another reason why OAuth 2 is a good choice as protocol is, that it is easily extensible to support other tasks than authentication, as it is an authorization protocol. So in the future the server where the authentication prototype is hosted, could support other tasks where a qualified identity is required.

For the architecture of the prototype inside the scope of OAuth 2 had to be made, as the specification is relatively open for interpretation. For this decisions the use-cases of the prototype are very important and are emphasized in this context.

4.1 Components of the prototype

As the architecture of the prototype includes many distributed components, they are visualized it in figure 4.1 for better understanding.

It represents an overview of the components and how they are related to each other. Some of them are discussed in section 2.5.2, when OAuth 2 was explained. Details regarding the implementation and structure are found in the upcoming chapter 5.

4.2 Use Cases

The main use case for the prototype is the authentication process. A service provider should be easily able to integrate the interaction to the prototype's interface in its client. This way they can be sure to only have users with qualified identities logged in to their services.

There is no need to support multiple identity providers or flexible entity attributes at this time, as the attributes available to us are already limited. Therefore it does not make

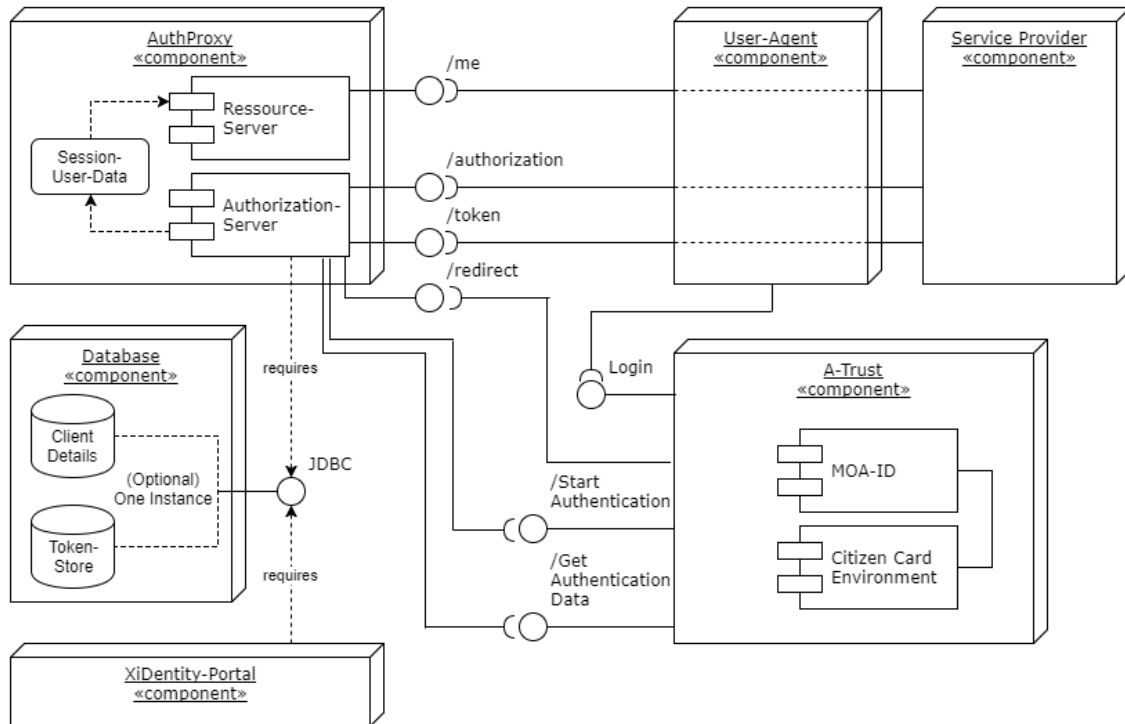


Figure 4.1: Components of the Prototype

sense to constrain them any further. For the same reason the OAuth 2 scope will not be used, as there is only the authentication use case present at this point in time.

For further extension the scope parameter can easily be activated and the prototype can be expanded for additional use cases.

4.3 Framework decisions

We will discuss certain decisions that were made in the process of planning the implementation of the prototype.

4.3.1 Token

In OAuth 2 there is the decision to make, which kind of access token is used for accessing resources - the so called 'Authenticator'. In the OAuth 2 core specification there is no particular token type specified [Har12]. Therefore in this section we will discuss the decision, which type of token to use in the prototypical implementation.

4.3.1.1 Bearer Token

Bearer Token is the most used form of token regarding OAuth 2. If no specific type of token is specified, most implementations use bearer token as default. This type of token simply means, that the bearer of it has the right to access resources owned by

the authenticated user, who has received the token. There is no need for any further cryptographic operations, therefore it is necessary to transport the token over a secure channel. Here is an example of a bearer token used by the Spring Security framework:

```
{
  "access_token": "ab12c345-d6ef-7891-23gh-45ij6k7l89mn",
  "token_type": "bearer",
  "refresh_token": "98z7y654-3xwv-2ut1-9876-54sr3qp2198o",
  "expires_in": 3600,
  "scope": "read"
}
```

So one of the reasons, why OAuth 2 strongly relies on a secure connection over SSL/TLS is, that the plain `access_token` field can be used for further requests as it is transported in the HTTP `Authorization` request header or directly in the URL - in plain text. If transported over a insecure connection, the token could be intercepted during a man-in-the-middle attack and used for further requests to the resource server by the attacker. Also while storing the tokens as a client, it is crucial that they are kept secure, otherwise the same attack-scenario could be applied by getting the tokens from a unsecured database.

There are optional recommendations for the usage of bearer tokens in the official specification [JH12], as it is not very restrictive. So it is not recommended, to store the token inside cookies, as the default transmission mode for cookies is not secure. If nevertheless the decision is made to store the token in the cookies, precautions must be made to prevent cross-site request forgery.

As the access token can be transmitted electively via page URLs (e.g. as GET parameter: `https://[...]?access_token=ab12c345-d6ef-7891-23gh-45ij6k7l89mn`) or in the HTTP message itself, the client has to make this choice. The recommendation is, to transmit the access token in the `Authorization` field of the header, as opposed to passing it as URL parameter. Involved software could inadequately process these URLs (including the sensitive parameter) by i.e. storing it in the browser history or writing it to server logs, where the token could be leaked.

Following the principle of least privilege, the bearer token should also contain a scope and the lifetime should be rather short-lived. That means, that it should not be valid more than one hour and only valid for specified resources. These steps help to prevent the malicious impact, if the token should be leaked contrary to expectations.

4.3.1.2 Message Authentication Code (MAC) Token

There is also the possibility to use a MAC Token as token type. The advantage is, that when using mac tokens, no sensitive data has to be transported every time when requesting a resource from the resource server [Sir]. Additionally to the Bearer Token, the MAC Token has three more fields:


```

{
  "access_token": "ab12c345-d6ef-7891-23gh-45ij6k7l89mn",
  "token_type": "mac",
  "refresh_token": "98z7y654-3xwv-2ut1-9876-54sr3qp2198o",
  "expires_in": 3600,
  "scope": "read",
  "kid": "0s9jdfAMLD/Aa0j8s9S0F=",
  "mac_key": "394tvn4zn34ct7",
  "mac_algorithm": "hmac-sha-256"
}

```

For the MAC Token there is no complete standardized specification yet, but a expired draft in its fifth version exists from 2014 [RMTH14] (work in progress) which also has recommendations for the usage. The new parameters are explained there as following:

kid The key id (kid) is used as an identifier. It is recommended, that it is computed by hashing over the `access_token` and encoding it in BASE64.

mac_key This key is a session key, which is created by the authorization server. The lifetime of the `access_token`, which is specified in `expires_in` applies also to the `mac_key`. This parameter is to be held secret, as it is used to sign further requests.

mac_algorithm As the name suggests, this parameter specifies the MAC algorithm which is used to calculate the MAC for further requests.

The data which has to be send each request by the client contains more parameters than for the bearer token, where it is just the `access_token`. For mac tokens following parameters in the `Authorization` field are required:

Authorization:

```

kid="0s9jdfAMLD/Aa0j8s9S0F=",
ts="670593600",
mac="7h1515mY81r7HDAy="

```

There are more optional parameters, which are left out in this scope. The presented parameters above are sufficient, to make a valid request to the resource server. The mentioned parameters are computed the following way:

kid The key id (kid) that was obtained by the initial MAC Token, to map the request to the user.

ts A (unix) timestamp of the current request.

mac The generated message authentication code (MAC) which is generated by using the defined `mac_algorithm` as algorithm, the `mac_key` as key and a predefined concatenation of several HTTP header fields as input.

The output of the signature algorithm is then again `base64Url`-encoded and concatenated to header and payload with another dot, which gives us our final JWT token. There exists a proposed standard, which is rather comprehensive [JCM15].

4.3.1.4 Own implementation

There is the possibility to define an own mechanism for issuing and using access tokens. We will not further look into this topic, because proprietary algorithms will only have security drawbacks and the clients won't be able to understand the tokens without an proprietary implementation on their side.

4.3.1.5 Conclusion

As for the prototype, the decision was made to use Bearer Tokens. This choice is justified by the ease of use and the compatibility among clients. This is an important argument, because of the requirement, that the service should be easy to implemented for the service providers who are using it.

Even if the MAC Token, as well as the JWT seem to have the same properties but advanced security features opposed to the Bearer Token, these features can be neglected [Gol]. There is no additional security benefit by using MAC Tokens over Bearer Tokens, because if SSL/TLS is used, which is mandatory for the OAuth 2 protocol, the attack vectors for both types are the same. If the SSL/TLS layer is compromised, the MAC Token would have an additional step towards security, but in total the whole protocol and therefore the prototype implementation would fail.

When using bearer tokens the clients must ensure the secure storage of the access tokens. One could argue, that this step is also a drawback when compared to mac tokens. But with mac tokens the clients also have to store the `mac_key`, they share with the server, so the secure storage infrastructure is needed anyway. Another important argument for bearer tokens is, that they can be easily revoked. For revoking a bearer token it just has to be deleted from the storage. When using tokens, that contain all the information inside the token itself, we don't store them and therefore the reference to this token can't be deleted. There is no standardized way of revoking these kind of tokens, but if controlling both, the resource- and authorization server there is a way to implement it. To achieve this, references to revoked tokens for the period of their validity have to be stored. This mechanism is not standardized, so it is another possible source for implementation mistakes.

In conclusion there are no major advantages by using any other token type than the standard bearer token in the implementation.

4.3.2 Token Store

Another design decision concerns the token store. Following possibilities exist, to choose from:

4.3.2.1 In-Memory Token Store

The `InMemoryTokenStore` is easily explained. The details of the client and other information (like created tokens) are stored in volatile memory of the server. The clients are statically configured before starting the server. So this kind of token store is inflexible, when we want to allow clients to dynamically add their API keys at runtime. Also it is hard to keep an overview over the currently existing access tokens.

When using distributed systems, there is no way to access the token store remotely from another server than the one, which has the token stored in memory. This makes it impossible to implement an architecture, with redundant servers. This also means, that when restarting the server, all information is lost without a possibility to back it up.

So this kind of token store is great for testing the implementation while developing, as there is no additional database or structure needed. In conclusion it is not meant to be used in a productive environment.

4.3.2.2 JDBC Token Store

The Java Database Connectivity (JDBC) token store is much more advanced and flexible than the `InMemoryTokenStore`. JDBC is an API for accessing databases, which means it is possible to use any relational database in the background. Thus the databases can be easily exchanged and hosted externally, which means that different server instances can access it.

As there is a predefined database schema for the Spring OAuth 2 implementation clients can be manually added at runtime. The process of client creation can also be automated via an interface.

The schema includes, among others, following tables [Sye]:

oauth_client_details contains all the information for a client. This includes the API key and secret, the redirect URL and other important information.

oauth_access_token stores the actual access tokens of the users.

oauth_refresh_token holds the refresh tokens (if configured) of the users, with which the user can request another access token after expiration of the same.

oauth_approvals contains the user approvals. Every time a client requests a resource of a user from the resource server, the user is asked (if the field `autoapprove` is not activated in `client_details`), if they grant the client permission to this resource. This approval is then stored for later reference.

4.3.2.3 JWT Token Store

As explained in subsection 4.3.1 there is no specific store in the backend for Java Web Tokens, as the information is encoded directly in the token. So when choosing this token store, Spring stores the information directly in the tokens itself.

This is not an option for the implementation of the prototype, as we chose to use Bearer Token as token type, which is incompatible with the `JwtTokenStore`.

4.3.2.4 Conclusion

For the implementation of the prototype we chose to use the `JdbcTokenstore`. There is no real alternative to it, as we need the advanced functionality a external token store provides.

4.3.3 User ID Obfuscation

When users register at a website or a service through our prototype, they are identified by a unique ID. A user always has the same ID when authenticating via our service. When this ID now gets passed to the clients without any obfuscation, the users would be trackable across providers like visualized in figure 4.2.

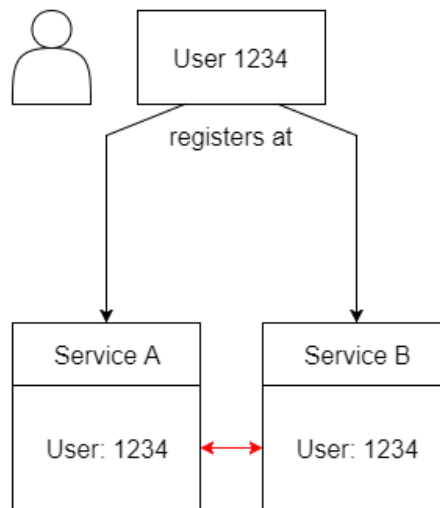


Figure 4.2: Passing the user ID to the different clients without obfuscation

For privacy reasons there should be a mechanism in place, that obfuscates the ID of the user for different clients. This obfuscation can be achieved by using a cryptographic hash function like SHA-256. As input for this one-way function we need three parameters:

- User ID
- Client ID
- Constant

This way we can assure, that the user ID stays consistent for every client. The constant, is hard-coded and specific for the instance of the prototype. Thus different instances can provide different IDs for the users, which is useful when e.g. offering a testing environment for developers. With this added obfuscation step, the linking of the same user between different clients is prevented, as seen in figure 4.3.

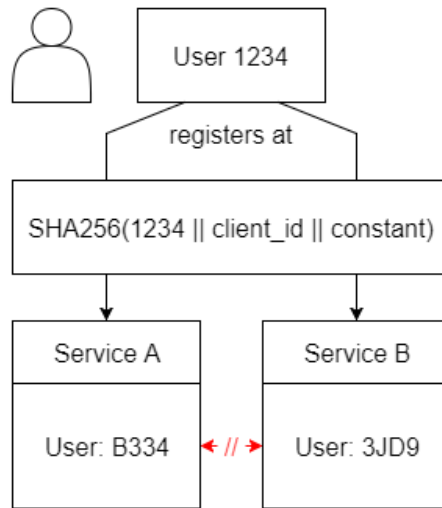


Figure 4.3: Passing the user ID to the different clients with obfuscation

Chapter 5

Experimental Evaluation

In this chapter the implementation of the actual prototype is discussed. At first, a version of the prototype was created where different approaches were tested and reviewed for feasibility. This first project had no claim to be exhaustive and was used as proof of concept.

The second prototype only contains the certain parts of the first prototype, which were proven to be beneficial. This implementation is structured and contains all the pieces for deployment in a testing environment.

For testing purposes the server runs locally on the development machine and for simplicity it does not use SSL/TLS for a secure connection. In an productive environment the use of a secure connection is indispensable. In the following chapter the implementation process is divided into the different stages of the authentication process.

5.1 User Authentication

First of all the user who uses the service has to be authenticated. Thus for authenticating the users with their qualified identity, a connection between our API and an identity provider has to be established. In this case this identity provider is a citizen card server (BKS), that has access to registered XiDentity users. A-Trust provided us access to their test environment 'a.sign BKS' which is reachable via the URL

```
https://test1.a-trust.at/
```

including a documentation [Hag15], based on which most of the following steps are performed.

5.1.1 Obtaining the SAML-Artifact

The communication between our API and this citizen card server is crucial for the implementation, because in this part we obtain the qualified identity of the users who want to authenticate themselves via our API as seen in figure 5.1.

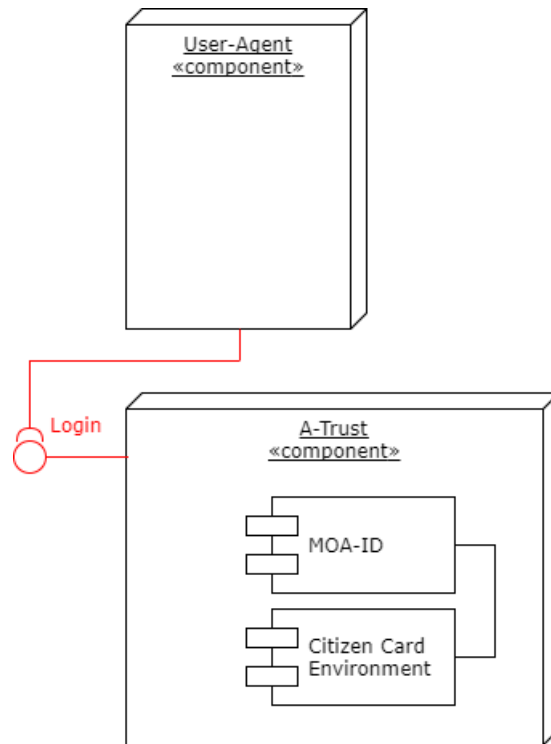


Figure 5.1: Login

For an authentication with the BKS a connection to the MOA interface (for more information see section 2.5.7) has to be established. Two variants exist for connecting to the interface:

1. Authentication via browser forwarding
2. Authentication without browser forwarding

Authentication via browser forwarding is the faster way to implement the functionality, but the flexibility to adapt the design of the login form is not given. As a first step and for a fast result this method is implemented, to test if everything works as it should. With this method the website of the BKS is shown to the user in the browser. To reach the server the connection to

```
https://<server>/ASignBuergerkartenServer/
StartAuthentication.aspx?OA=<url>
```

has to be made, where **OA** stands for 'online application'. With this parameter the BKS can be notified where to redirect the user's browser after the successful authentication process. The MOA server first has to register the URL of the online application, which is used to perform the authentications. In our case this parameter is

```
http://xitest.ddns.net:8096/MoaRedirect/
```


which, in combination with the BKS URL leads to the compound URL

```
https://test1.a-trust.at/assignBuergerkartenServer/StartAuthentication.aspx?
OA=http://xitest.ddns.net:8096/MoaRedirect/
```

which corresponds to the connection between the components as illustrated in figure 5.2.

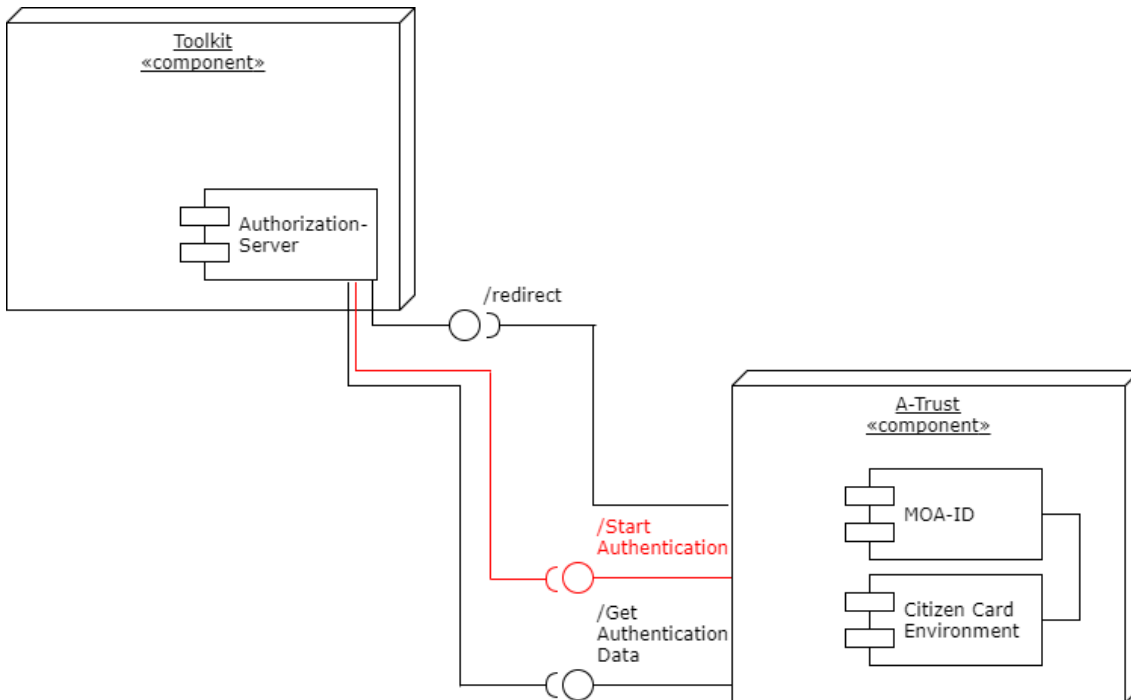


Figure 5.2: StartAuthentication interface between components

By implementing this HTTP-GET request, represented by the URL, into a sample application and showing the resulting page to the user, it was possible to reach the first page of the authentication process - the standard template as seen in figure 5.3.



Figure 5.3: Standard template of the first step in MOA authentication step

After clicking the Login button, the A-Trust login prompt was supposed to be displayed. Unfortunately the connection is not established as the browser tries to connect to the local address

```
https://127.0.0.1:3496/https-security-layer-request
```

which refuses the connection, because we don't have a Bürgerkartenumgebung running on the local development machine. As stated in the documentation, there are further optional parameters for the initial authentication request which can be passed. One of them is the parameter `BKU=<bku>`, where `<bku>` can be either `local` for a locally installed BKU, `online` for a BKU that is reachable online or `handy` for login via Handy-Signatur.

As we want to use Handy-Signatur for our authentication mechanism `BKU=handy` is appended to our URL which now looks like follows:

```
https://test1.a-trust.at/asignBuergerkartenServer/StartAuthentication.aspx?
OA=http://xitest.ddns.net:8096/MoaRedirect/&BKU=handy
```

With this approach the login prompt is successfully displayed, which is illustrated in figure 5.4.



Figure 5.4: Login prompt for Handy-Signatur

Now when entering valid credentials the system is expected to be forward the browser to the next step. This next step would be a TAN based two factor authentication mechanism of the A-Trust login. After entering a valid phone number and the signature password, an error occurs. This error predicates, that the entered credentials are wrong. After inspection of the browser's redirection address, this time no a local address is found, but the following:

```
https://test1.a-trust.at/mobile/https-security-layer-request/
identification.aspx
```

When checking back with A-Trust, the instructions were received that in that case the URL to the desired BKU has to be entered explicitly. This is done through the parameter `BKUUrl=<url>`, which overrides `BKU=<bku>`. So the BKU URL that corresponds to the proper Handy-Signatur service is requested, which is located at

```
https://www.handy-signatur.at/mobile/https-security-layer-request/
default.aspx
```

With all components combined the final URL is merged, which looks like this:

`https://test1.a-trust.at/assignBuergerkartenServer/StartAuthentication.aspx?OA=http://xitest.ddns.net:8096/MoaRedirect/&BKUUrl=https://www.handy-signatur.at/mobile/https-security-layer-request/default.aspx`

After entering this new URL as destination in the demo application, the login prompt visualized in figure 5.4 is displayed again. This time the entered credentials are accepted. Thus the browser is redirected to the next step of the login, which is the TAN mechanism as shown in figure 5.5.

Figure 5.5: TAN prompt for Handy-Signatur

The TAN is entered, after comparing the reference value from the prompt with the one in the SMS, which was received immediately after the prompt is shown. Subsequently the browser is successfully redirected to the specified URL, that was passed as the `OA` parameter.

The core functionality is working as it should. As subsequent step, the variant where the first step regarding browser forwarding is skipped.

Authentication without browser forwarding is the way, the MOA-Login should be integrated to our API. For the integration of this mechanism, one additional step is needed at the beginning of the process. For the first version of the prototype a dynamic HTML form is used, which is filled with values retrieved via a REST call to the familiar URL

`https://test1.a-trust.at/assignBuergerkartenServer/StartAuthentication.aspx?OA=http://xitest.ddns.net:8096/MoaRedirect/&Format=JSON`

At the end of the URL the parameter `Format=JSON` is appended. This signalsizes, that the data needed for our HTML form as a JSON response is expected. The dynamic HTML form is filled with the following values

```
<form action="{BKUUrl}" enctype="application/x-www-form-urlencoded"
  method="post" name="{formName}">
  <input type="hidden" name="XMLRequest" value="{XMLRequest}" />
  <input type="hidden" name="DataURL" value="{DataURL}" />
  <input type="submit" text="Start_Login" />
</form>
```

For the second version of the prototype the HTML representation of the form is skipped and the needed data is directly sent to the server via a HTTP POST request. This way the user is automatically redirected to the login prompt without needing to click any button.

After an successful authentication, the redirect to the specified URL happens in which also the HTTP GET parameter `SAMLArtifact` is received. This value is used to obtain the data of the authenticated user. To get this data it is necessary to communicate with yet another interface - a SOAP interface.

5.1.2 Getting the SAML Assertion via the SOAP Interface

For further communication it is necessary to obtain the SAML Assertion, which can be achieved by sending the received SAML Artifact via the SOAP interface, which is located at

`https://test1.a-trust.at/assignBuergerkartenServer/GetAuthenticationData.asmx`

The first step for communicating with the SOAP Service is, to get the WSDL file, by attaching the parameter `WSDL` to the above URL as seen in figure 5.6.

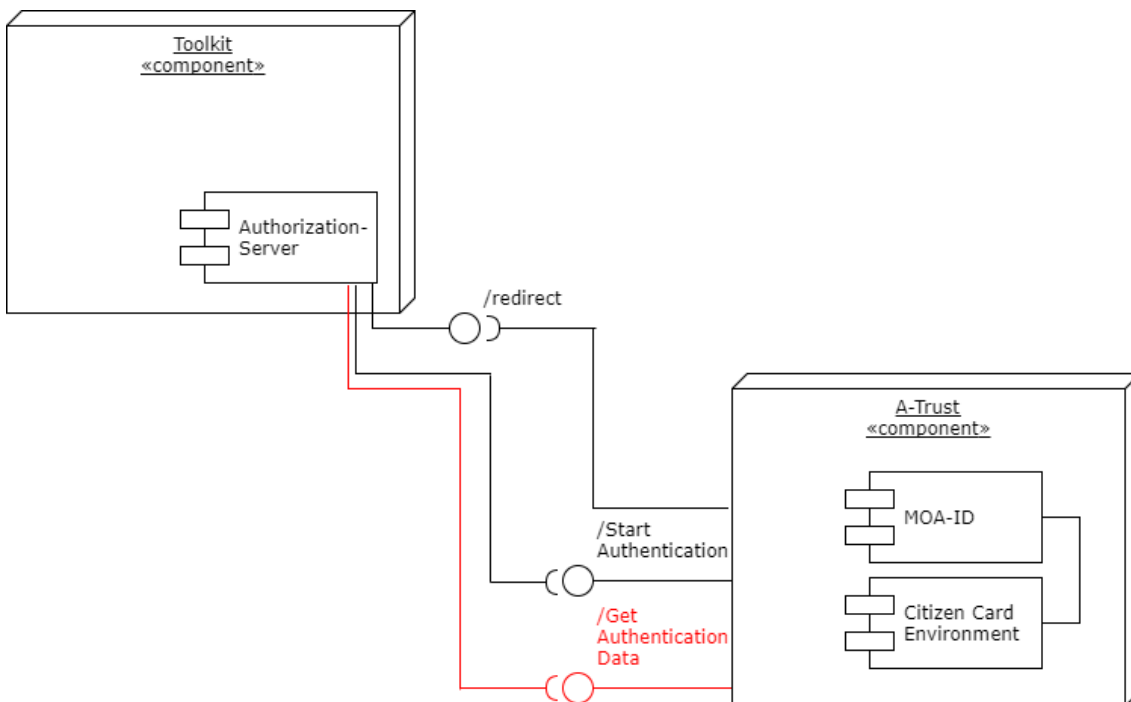


Figure 5.6: GetAuthenticationData between components

With this file as input for the `maven-jaxb2-plugin`¹ the Java classes are created, which are needed for the following step. This process has to be performed only once before runtime.

¹<https://java.net/projects/maven-jaxb2-plugin/pages/Home>

Now, by sending the SAML-Artifact, which was obtained in step 5.1.1 to the available `GetData` operation available at

`http://www.a-trust.at/aSignBuergerkartenServer/GetData`

a XML response is received as illustrated in figure 5.7. This XML response is, if the previous steps were successful, a valid SAML-Assertion that contains the data of the authenticated user.

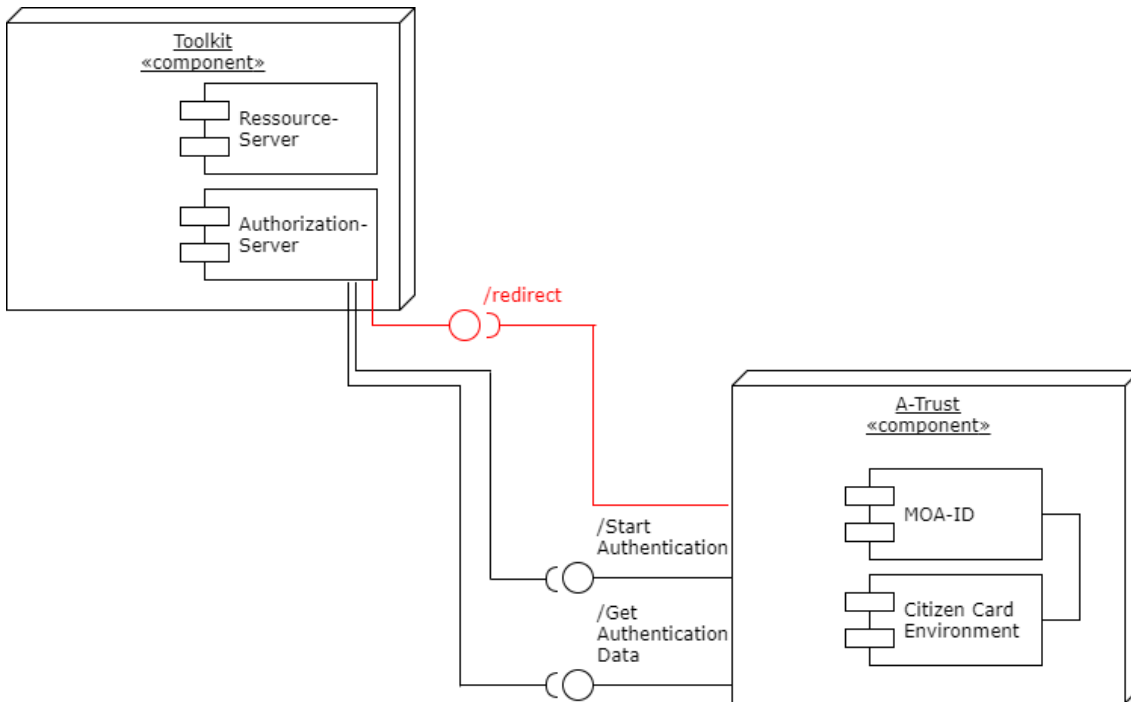


Figure 5.7: SAML response to redirect URL

5.1.3 Processing the SAML-Assertion

After successfully obtaining the SAML-Assertion, there is a lot of information to process. There are two ways, how a user can currently use the MOA login: via Handy-Signatur and via XiDentity. Dependent on the method used, the SAML-Assertion looks different. The information also has to be verified and checked for validity.

5.1.3.1 Parsing the user information

As mentioned above, there is a structural difference regarding the SAML-Assertion between the two methods of authentication.

The Handy-Signatur SAML-Assertion is containing the given name, the last name and the date of birth of the authenticated user. It also contains a unique id to identify the user to our service. This identifier does not change over time or after the certificate of the user is changed.

The XiDentity SAML-Assertion does not contain the same information as the Handy-Signatur SAML-Assertions. They are missing the date of birth of the user and as an id A-Trust provides the serial number of the attached certificate. This turned out to be a problem in the long run, because when revoking the certificate (i.e. when the user forgets the signature password) or extending the lifetime of it before it expires, the unique identifier of the users would change. Therefore the toolkit could no longer map the authenticated user to the old identity and the user would be recognized as completely new individual. This would lock the users out of all their accounts which are connected to the old id. In cooperation with A-Trust a concept was developed to solve this problem, which is currently implemented. The serial number of the first created certificate is used as unique id. Every further revocation or renewal of certificates is protocolled at the A-Trust data center. This way the new certificates can be mapped to the old identities.

By using and configuring a XML parser, the information for both assertion types can be extracted. Thus the defined tags inside the document have to be accessed, which include the needed information.

5.1.3.2 SAML-Assertion Validation

For additional security, certain aspects of the SAML-Assertion are checked and validated.

Inside the assertion, a X.509 certificate of the authenticated user is attached. After parsing the BASE64 encoded certificate, a check can be carried out, if the certificate is still in its validity range and therefore not expired. Also if the issuer of the certificate is valid, can be reviewed.

The whole SAML-Assertion is signed with the certificate's private key at the A-Trust trust center. We can verify, that the content was not subsequently modified by verifying the XML signature with the public key of the transmitted certificate.

5.2 OAuth 2 Flow: Communication between Client and API

After successfully authenticating the user at the prototype, the client can initialize the OAuth 2 flow. The endpoints which are used in this step are highlighted in figure 5.8.

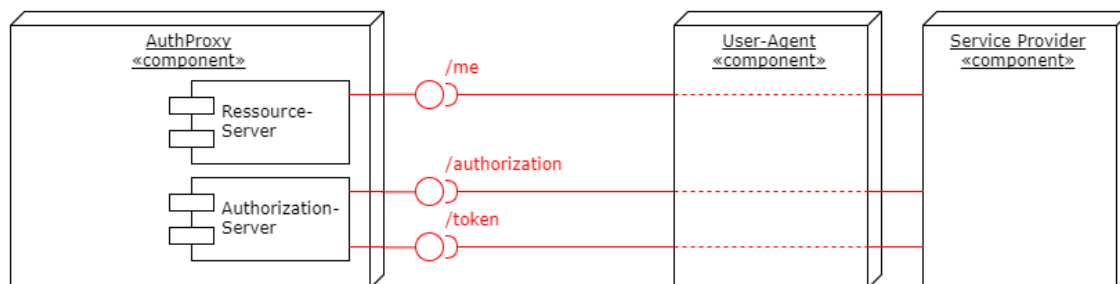


Figure 5.8: Endpoints used by clients

To validate the functionality of the prototype, two different clients are implemented to test the API. For the first client a Java implementation with the aid of the Spring-Boot-Framework is created. The other client is written in the script language PHP. As one requirement for the API was, that it should be easy to integrate in an environment, the creation of said lightweight clients is not very complex.

Additionally to the showcase clients the application 'Postman' will be introduced, as it helped to test the prototype during the implementation process.

As for special cases, where the API is integrated when there are already other authentication mechanisms in place, useful hints can be found in the Facebook Login documentation discussed in section 3.1.1.

5.2.1 Spring Client

For the first client, which is written in Java, the frameworks 'Spring Boot'² and 'Spring Security'³ are used. With the aid of the available built-in components it is fairly simple to create an OAuth 2 client.

With the Annotations `@SpringBootApplication` and `@EnableOAuth2Client` the project is specified as a Spring-Boot application that contains the functionality of an OAuth 2 client. Furthermore the client has to be configured with the following parameters

- `client_id`
- `client_secret`
- The authorization endpoint
- The token endpoint
- The user information endpoint

The configuration of these parameters is defined in the `application.yml` file, which looks like follows:

```

prototypelogin:
  client:
    clientId:          KA8NJSY5QKBPEWXW55IV5HU6PJC6XPM
    clientSecret:      ZXIJBHYZWQZS4WIJ55PWENBMU4QLU2GTC9JISCLEL
    accessTokenUri:    http://xitest.ddns.net:8096/auth/oauth/token
    userAuthorizationUri: http://xitest.ddns.net:8096/auth/oauth/
                        authorize
  resource:
    userInfoUri:      http://xitest.ddns.net:8096/auth/me

```

For the first step it is necessary, to be authenticated at the API. To achieve this, a redirect to the `/oauth/authorize` interface of the authorization server we specified in the configuration is needed. This can be accomplished, by registering a Spring-Security filter that handles the access management in our Spring client:

²<https://projects.spring.io/spring-boot/> (accessed July 26, 2017)

³<https://projects.spring.io/spring-security/> (accessed July 26, 2017)

```

@Override
//In this method we set the actual filter we create further down
protected void configure(HttpSecurity http) throws Exception {
    http.antMatcher("/**").authorizeRequests()
        .antMatchers("/", "/login**").permitAll().anyRequest()
        .authenticated()
        .authenticationEntryPoint(new LoginUrlAuthenticationEntryPoint("/"))
        .and()
        .addFilterBefore(oauthFilter(), BasicAuthenticationFilter.class);
}

@Bean
@ConfigurationProperties("prototypelogin")
//We use a Java Bean, to get the ClientResources with our configured data as
//configuration properties. These ClientResources consist of
//AuthorizationCodeResourceDetails and ResourceServerProperties
public ClientResources prototypelogin() {
    return new ClientResources();
}

//This method is called with the ClientResource object created via the above
//function prototypelogin() and the path the filter should be applied (e.g
//. "/login/prototypelogin")
private Filter oauthFilter(ClientResources client, String path) {
    //Creation of the OAuth2ClientAuthenticationProcessingFilter
    OAuth2ClientAuthenticationProcessingFilter
        oauth2ClientAuthenticationFilter = new
        OAuth2ClientAuthenticationProcessingFilter(path);
    //Initialize the OAuth2RestTemplate with the data from our
    //ClientResources object to make REST calls to the API
    OAuth2RestTemplate oauth2RestTemplate = new OAuth2RestTemplate(
        client.getClient(), oauth2ClientContext);
    //Connect the created OAuth2RestTemplate to the
    //OAuth2ClientAuthenticationFilter
    oauth2ClientAuthenticationFilter.setRestTemplate(oauth2RestTemplate);
    //Create the UserInfoTokenServices object we need to obtain the user
    //info data from the specified userInfoUri interface at the API
    UserInfoTokenServices tokenServices = new UserInfoTokenServices(
        client.getResource().getUserInfoUri(), client.getClient().
        getClientId());
    //Connect the OAuth2RestTemplate also to the UserInfoTokenServices
    tokenServices.setRestTemplate(oauth2RestTemplate);
    //Connect the created UserInfoTokenServices to the
    //OAuth2ClientAuthenticationProcessingFilter
    oauth2ClientAuthenticationFilter.setTokenServices(tokenServices);
    //Finally return the created filter
    return oauth2ClientAuthenticationFilter;
}

```

After defining that filter, the redirect to the login prompt is active. After a successful authentication, the user agent establishes the connection back to our client. The next steps are handled automatically, because the relevant class of our implementation is remarked with the `@EnableOAuth2Client` annotation:


```

@SpringBootApplication
@EnableOAuth2Client
public class SocialApplication extends WebSecurityConfigurerAdapter {
    [...]
}

```

So the client performs the request for the access token via `/oauth/token` as a next step and then sets the session authentication to an so called `OAuth2Authentication`. With this authentication in place it is possible to access the user info endpoint at `/me` and with these steps the authentication process is finished. As a result we are able to access the authentication data of the authenticated user by reading it from the `OAuth2Authentication`. Therefore we created a REST interface URL for the client at `/printuserdata`, which returns the mentioned data as a JSON file:

```

@RequestMapping("/{printuserdata"})
public Map<String, String> user(Principal principal) {
    //Get the principal and authentication of the authenticated user
    OAuth2Authentication oAuth2Authentication = (OAuth2Authentication)
        principal;
    Authentication authentication = oAuth2Authentication.
        getUserAuthentication();
    //Parse them as JSON
    Map<String, String> details = (Map<String, String>) authentication.
        getDetails();
    JSONObject details_as_json = new JSONObject(details);
    map.put("id", details_as_json.getString("id"));
    map.put("firstname", details_as_json.getString("firstname"));
    map.put("lastname", details_as_json.getString("lastname"));
    map.put("dob", details_as_json.getString("dob"));
    return map;
}

```

5.2.2 PHP Client

For the PHP client we could implement a simple client by ourselves. In practice developers would utilize external libraries, as they are often more extensive and already implement convenient tasks like session- and exception handling. Thus to simulate the usage of our API in a real world scenario, we rely on a external OAuth 2 client library under MIT licensing⁴.

The first step is the same as for the Spring client. We need to configure the client details and endpoint URLs. As we can see in figure 5.9, the minimal UI shows us the configured client details:

⁴https://github.com/vznet/oauth_2.0_client_php (accessed July 26, 2017)

Consumer Key: 3P2K2[redacted]
 Consumer Secret: WUX[redacted]
 Access Token: [redacted]
 Refresh Token: [redacted]
 LifeTime: [redacted]

[authorize](#)

[request API](#)

Figure 5.9: Minimal UI of the PHP client

After clicking on the `authorize` button, the client redirects the user to the known login prompt (as already visualized in figures 5.4 and 5.5). Following the OAuth 2 flow, the redirect back to the client happens after a successful authentication and the `/oauth/token` interface is called with the according parameters. Now the missing data is automatically filled into the blank UI fields. The last step missing is to access the `/me` interface of the API. This is achieved by clicking the `request API` button. As seen in figure 5.10 the request returned the data of the authenticated user as expected:

```
{"id":"2TRT5[redacted]","firstname":"[redacted]","lastname":"[redacted]","dob":"19[redacted]"}
```

Consumer Key: 3P2K2[redacted]
 Consumer Secret: WUX[redacted]
 Access Token: 5e3158a5-[redacted]
 Refresh Token: 5896539f-[redacted]
 LifeTime: 1506591773

[authorize](#)

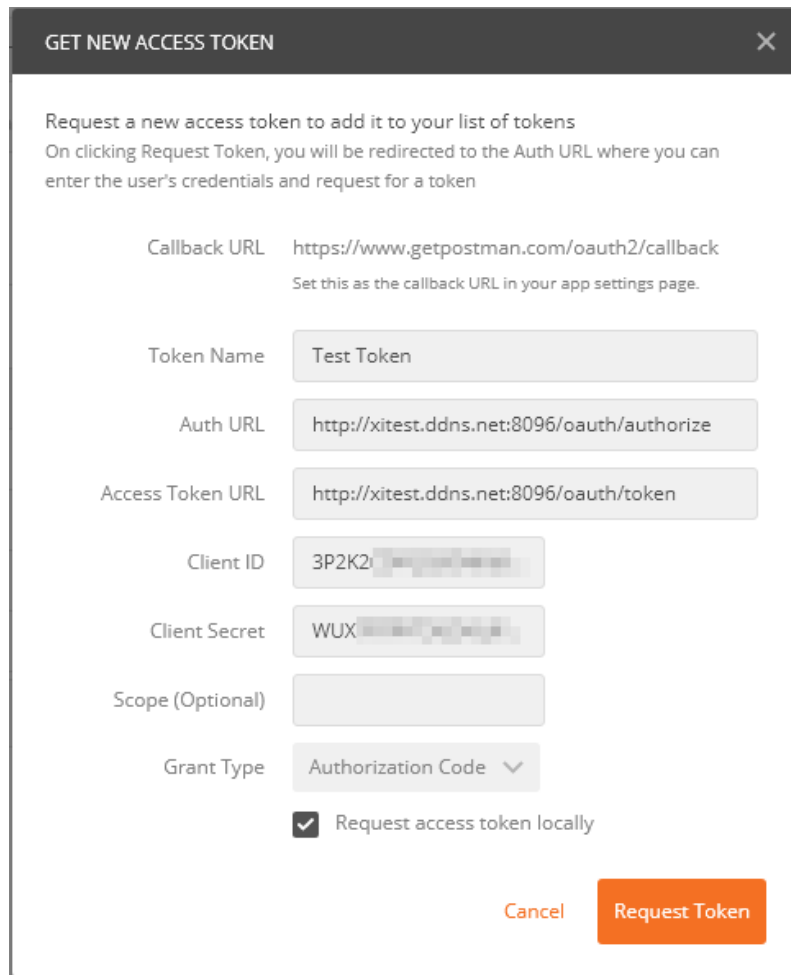
[request API](#)

Figure 5.10: User data is returned and blank UI fields are filled

After this last step the authentication process is concluded. This shows, how easy the integration of OAuth 2 is and how fast it is to include in an application.

5.2.3 Postman

Postman⁵ is an application, for developing and testing APIs. It was frequently used during prototyping and has a build-in OAuth 2 authentication method. When requesting tokens via this method, a prompt is shown, that can be seen in figure 5.11. Via this module a full OAuth 2 flow can be executed.



GET NEW ACCESS TOKEN

Request a new access token to add it to your list of tokens
On clicking Request Token, you will be redirected to the Auth URL where you can enter the user's credentials and request for a token

Callback URL `https://www.getpostman.com/oauth2/callback`
Set this as the callback URL in your app settings page.

Token Name

Auth URL

Access Token URL

Client ID

Client Secret

Scope (Optional)

Grant Type

Request access token locally

Cancel Request Token

Figure 5.11: Getting new access token via Postman

After connecting to the authorization endpoint of the prototype the user is prompted with a web-view containing the login prompt for the user. Following the login the application is contacting the token endpoint with the gained authorization code and exchanging it against an access token. When the flow was successful, the newly gained access token is stored manually and available to the user. It now can be decided, if the token should be used in the header of a request to the resource server or as URL parameter. If used directly in the URL, Postman appends the access token to the resource server URL the following way:

⁵<https://www.getpostman.com/apps> (accessed September 11, 2017)

```
http://xitest.ddns.net:8096/me?access_token=44553eb5-f438-4a74-9162-502457050d23
```

When choosing 'Add token to Header', the token is added to the HTTP Header in the request to the resource server. In plain text it looks like this:

```
Authorization:Bearer 44553eb5-f438-4a74-9162-502457050d23
```

As we learned in section 4.3.1.1, in productive environments the second method should be preferred.

This fast, transparent and dynamic way of testing the API while in development helped, to locate errors, with the certainty that the error is not part of a faulty client implementation.

Chapter 6

Conclusion And Future Work

In this last chapter we will discuss the outcome of the thesis and further steps that can be made in order to extend the prototype or make it ready for production.

6.1 Conclusion

The initial requirements were fully met by the second version of the implemented prototype. In cooperation with the company XiTrust the creation was performed from scratch and resulted in an operational application which already is in use for proof of concept showcases. First of all the requirements were evaluated and an analysis of the current situation in the field of authentication mechanisms was performed. Based on this analysis and in line with the requirements and use-cases the protocols and frameworks that are eligible for an implementation were chosen.

The architecture and details for the prototype were defined and the decisions made in this process were used to implement a first prototype. This implementation showed, that the authentication process could be realized with the chosen architecture. However, there were parts of the implementation, that were unnecessary and the configuration was more complicated than it needed to be.

Based on this first prototype and the lessons learned from implementing it, further development started to improve it. The second prototype was also started from scratch. This way only the necessary components were included into this new implementation. The configuration is user friendly and well documented.

With a test instance in place, the service can already be used with client side implementations.

6.2 Future Work

As the prototype is technically finished, it can be used as foundation for a future product or further research purposes. However as it was build to be flexible and easily expandable, there are multiple options for extending it as we will learn in the following sections.

6.2.1 Extend to OpenID Connect

By using OAuth 2 as protocol for the prototype, the system is open for various enhancements.

As we learned in chapter 4 we did not use OpenID Connect because of usability and distribution reasons. If the tradeoff from usability to security is desired, the possibility exists to extend the OAuth 2 base framework for an additional OpenID Connect layer. This enhances the security a small step further. This would mean, that the bearer tokens could be exchanged for JWT tokens and the client would have to perform signature validations. Also additional ID tokens would be in place.

6.2.2 Extend the API for additional tasks

Clients which use the service of the prototype can offer their users a secure way of authenticating themselves. As the prototype is based on OAuth 2 there is also a authorization protocol in place. Thus it has all the prerequisites to implement other tasks that authenticated users could benefit from. To accomplish this, the user-role- and scope-management has to be defined and new interfaces have to be created for various additional tasks. The extension of the API is visualized in figure 6.1. These tasks could be offered without implementing another interface in the client.

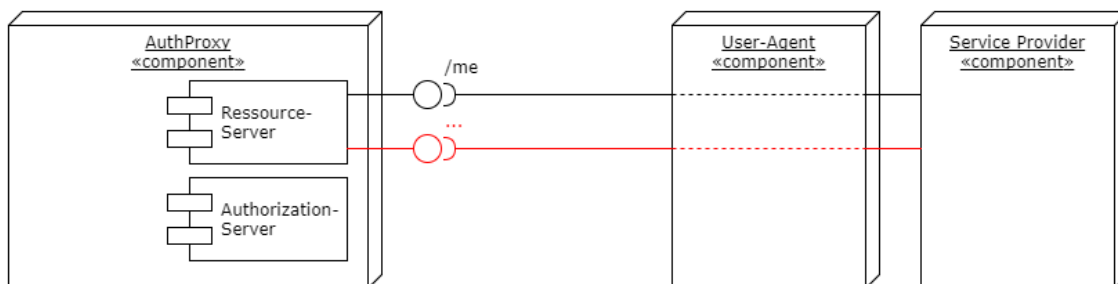


Figure 6.1: Extending the prototype for additional tasks

6.2.3 Legal audit

The prototype is a technical proof of concept of how the authentication via a qualified identity can be easily integrated in applications. The legal situation was taken in consideration while planning the prototype but has not been revised by experts in this field. Before the authentication service is used in an productive environment, the framework should be reviewed by legal professionals.

Also in respect to the upcoming EU regulation 2016/679, on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation) [PtCotEUa], the legal framework should be evaluated on further prototypes or products.

6.2.4 Security audit

In chapter 3 we got to know attack vectors regarding the prototype and OAuth 2 in general. These attack vectors often correspond to client-side implementations and responsibilities, therefore they cannot be prevented on the server side implementation level. A way to mitigate them is to educate developers how to implement and use the service. The place for documentation which helps developers with these issues is the developer portal, which will be mentioned in the next section.

When it comes to server-side implementation issues, they were taken into account while implementing the prototype. Not every issue is known at the moment of planing and implementing, so there may exists unknown weaknesses. An security audit should be performed to guarantee the security for the prototype in a broad area of application. These security audits often happen in the form of penetration tests, to find weak spots in the implementation.

6.2.5 Developer Portal

As an important step for using the prototype as authentication provider, it is necessary to provide a convenient developer portal for customers. This developer portal is integrated into the system as seen in figure 6.2.

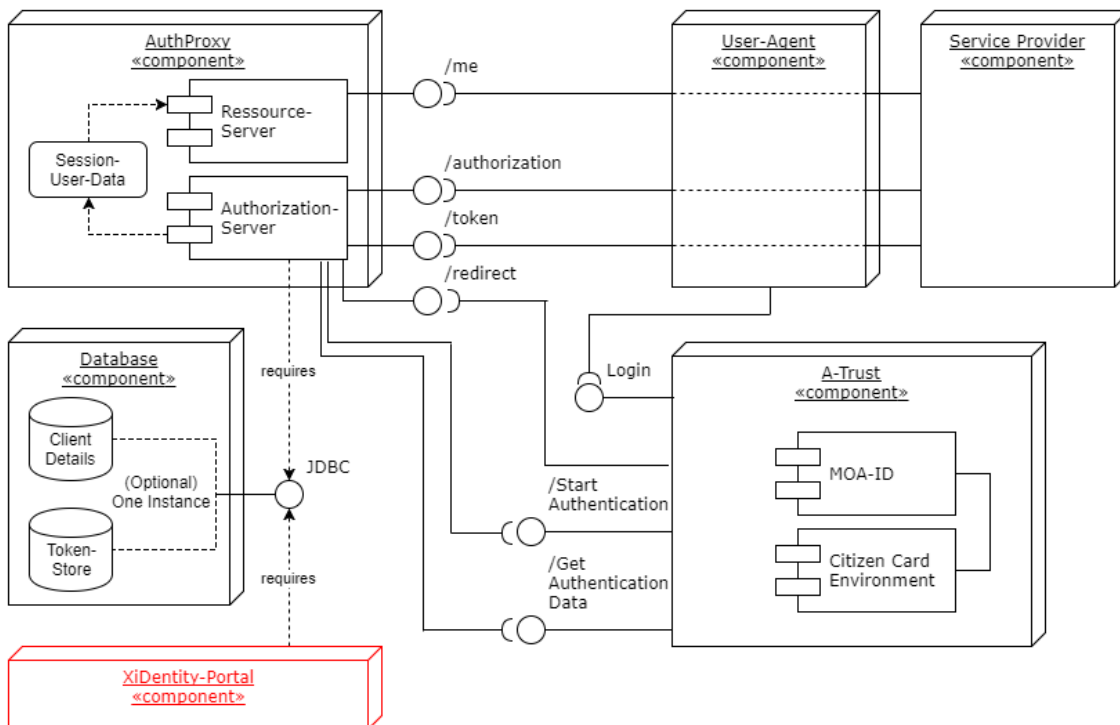


Figure 6.2: Developer portal in context

As for now, it is not possible to maintain registered clients or change data in a convenient way. The clients can be created and deleted via a REST-API. These two available

commands show, how to handle the clients. Thus a foundation for the developer portal can be created by adding further commands to said REST-API. Additionally it will be necessary to offer an appealing front-end to the customers, which communicates with this REST-API. That way they will be able to easily maintain their applications and the corresponding data like API keys and other details.

Nomenclature

- 2FA Two Factor Authentication, page 6
- API Application programming interface, page 2
- BKS Bürgerkarten Server (citizen card server), page 32
- eIDAS electronic IDentification, Authentication and trust Services, page 5
- ERnP Ergänzungsregister für natürliche Personen (Supplementary Register for Natural Persons), page 1
- IAM Identity and Access Management, page 1
- ID-FF Identity Federation Framework, page 13
- IdP Identity provider, page 5
- JDBC Java database connectivity , page 29
- JWT Java web token, page 14
- MAC Message authentication code, page 25
- MOA Module for Online Applications, page 15
- OA Online application, page 33
- PGP Pretty Good Privacy, page 6
- REST Representational state transfer , page 36
- RO Registration officer, page 5
- SAML Security Assertion Markup Language, page 13
- SOAP Simple Object Access Protocol, page 37
- SSL Secure sockets layer, page 7
- SSO Single sign on, page 13
- TLS Transport layer security, page 7
- UI User interface, page 42

WSDL Web services description language, page 37

XBS XiTrust Business Server, page 4

ZMR Zentrales Melderegister (Central Register of Residents), page 1

Bibliography

- [BKE14] Seymour Bosworth, M. E. Kabay, and Whyne E. *Computer Security Handbook*, volume 1. John Wiley & Sons, Incorporated, New York, USA, 6th edition, 2014.
- [Cen] A-SIT Secure Information Technology Center. Faq about the mobile phone signature. Online: <https://www.buergerkarte.at/en/faq-mobile.html> (accessed May 24, 2017).
- [dACA⁺14] Norberto Nuno Gomes de Andrade, Lisha Chen-Wilson, David Argles, Michele Schiano di Zenise, and Gary Wills. *Electronic Identity*. Springer Briefs in Cybersecurity. Springer, 2014.
- [dB12] Duncan de Borde. Two-factor authentication. *Insight Consulting*, 2012.
- [EGICEI] TU-Graz E-Government Innovation Center EGIZ IAIK. Moa-id. Online: <https://www.egiz.gv.at/en/schwerpunkte/13-moaspssid> (accessed March 10, 2017).
- [Fac] Facebook. Add facebook login to your app or website. Online: <https://developers.facebook.com/docs/facebook-login/> (accessed September 04, 2017).
- [Foua] OpenID Foundation. Specifications. Online: <https://openid.net/developers/specs/> (accessed August 11, 2017).
- [Foub] OpenID Foundation. Welcome to openid connect. Online: <https://openid.net/connect/> (accessed March 08, 2017).
- [Gmb] XiTrust Secure Technologies GmbH. Xitrust secure technologies gmbh. Online: <http://www.xitrust.com/xitrust-secure-technologies-gmbh> (accessed February 20, 2017).
- [Gol] Rahul Golwalkar. Pros and cons in using jwt (json web tokens). Online: <https://medium.com/@rahulgolwalkar/pros-and-cons-in-using-jwt-json-web-tokens-196ac6d41fb4> (accessed October 08, 2017).
- [Hag15] Patrick Hagelkruys. *a.sign Bürgerkarten Server*. A-Trust Gesellschaft für Sicherheitssysteme, The address of the publisher, 0.3 edition, 11 2015.

- [Har12] D. Hardt. The oauth 2.0 authorization framework. RFC 6749, RFC Editor, October 2012.
- [HL10] E. Hammer-Lahav. The oauth 1.0 protocol. RFC 5849, RFC Editor, April 2010.
- [JCM15] M. Jones, B. Campbell, and C. Mortimore. JSON web token (JWT) profile for OAuth 2.0 client authentication and authorization grants. Technical report, May 2015.
- [JH12] M. Jones and D. Hardt. The OAuth 2.0 authorization framework: Bearer token usage. Technical report, RFC Editor, October 2012.
- [JNN08] Anil K. Jain, Karthik Nandakumar, and Abhishek Nagar. Biometric template security. *EURASIP J. Adv. Signal Process*, 2008:113:1–113:17, January 2008.
- [LMH13] T. Lodderstedt, M. McGloin, and P. Hunt. OAuth 2.0 threat model and security considerations. RFC 6819, RFC Editor, January 2013.
- [Pey16] Sebastin E. Peyrott. *The JWT Handbook*. Auth0 Inc., 10900 NE 8th Street, Suite 700, Bellevue, WA 98004, 0.11.0 edition, 2016.
- [Por] Statista The Statistics Portal. Number of monthly active facebook users worldwide as of 2nd quarter 2017 (in millions). Online: <https://www.statista.com/statistics/264810/number-of-monthly-active-facebook-users-worldwide/> (accessed September 04, 2017).
- [PtCotEUa] The European Parliament and the Council of the European Union. Regulation (eu) no 2016/679 of the european parliament and of the council of 27 april 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing directive 95/46/ec (general data protection regulation) on electronic identification and trust services for electronic transactions in the internal market and repealing directive 1999/93/ec.
- [PtCotEUb] The European Parliament and the Council of the European Union. Regulation (eu) no 910/2014 of the european parliament and of the council of 23 july 2014 on electronic identification and trust services for electronic transactions in the internal market and repealing directive 1999/93/ec.
- [RHPM06] Nick Ragouzis, John Hughes, Rob Philpott, and Eve Maler. Security assertion markup language (saml) v2.0 technical overview. Technical report, October 2006.
- [RMTH14] J. Richer, W. Mills, H. Tschofenig, and P. Hunt. OAuth 2.0 message authentication code (mac) tokens (work in progress). Technical report, jul 2014.
- [Sir] Prabath Siriwardena. OAuth 2.0 bearer token profile vs mac token profile. Online: <https://dzone.com/articles/oauth-20-bearer-token-profile> (accessed October 08, 2017).

- [Sye] Dave Syer. `spring-security-oauth`. Online: <https://github.com/spring-projects/spring-security-oauth/blob/master/spring-security-oauth2/src/test/resources/schema.sql> (accessed October 08, 2017).
- [Twi] Inc. Twitter. Authentication - send secure authorized requests to the twitter api. Online: <https://dev.twitter.com/oauth> (accessed August 11, 2017).
- [Zim91] Phil Zimmermann. *An Introduction to Cryptography*, chapter Phil Zimmermann on PGP, page 54. Network Associates, Inc., 1991.