univention
be open.

ID Broker architecture

Univention GmbH

May 16, 2024

CONTENTS:

CHAPTER

ONE

INTRODUCTION

1.1 About this document

- Target audience of the document are persons with technical skills; focus is on Univention Developers, Univention Operations and (partly) Service Providers.
- Knowledge about UCS and UCS@school is a prerequisite.
- The document is updated on a regular base to follow the current state of the implementation.
- In a future release the Univention ID Broker will include a process to revert pseudonymization service. This is not yet part of the architecture document.
- Security related questions will be reviewed in each chapter and not separately.

1.2 Big Picture - what is the Univention ID Broker?

The main objective of the Univention ID Broker is to ease the integration between identities of learners and teachers managed by school authorities or federal states and the various service providers for educational purposes with respect to the data protection regulations in Europe.

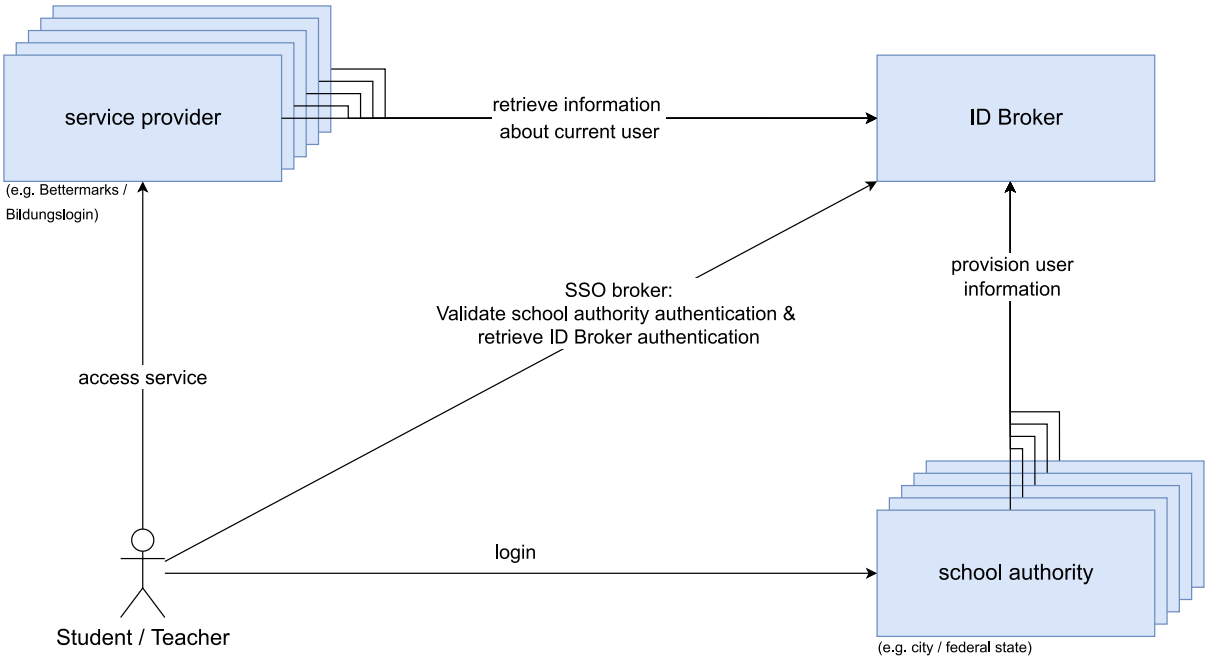


Fig. 1.1: Overview of the involved components of the ID Broker and external Systems.

To reach this goal the service will ensure:

- Single sign-on for end users between the identity provider (IDP) of a school authority / federal state (in the document summarized under the term *school authority*) and service providers (educational SaaS offerings).
- Only one configuration step to connect with the ID Broker both for IDPs and service providers (there is no need to configure each IDP with each service).
- Service specific pseudonyms instead of global identifiers for users to ensure that an users activities in the different services can't be combined to a "profile" of the user.
- To give end users a "complete" environment from scratch, service providers can retrieve information about the role and the courses of users.
- For services which must not get access to clear text information about a user, the ID Broker will provide (de-)pseudonymization services.
- To ensure data protection, the ID Broker environment will store only a minimal data set.

1.3 Use Cases

1.3.1 Overview

The list in this document describes the *high level use cases* for end users, service providers and administrators of identity providers.

1.3.2 End user single sign-on

As end user of an school authority, I'm authenticated in the environment of my school authority (typically by logging in to the Univention Portal hosted by my school authority). I want to access a SaaS without entering login & password, but based on a secure single sign-on.

1.3.3 End user comfort in SaaS offering

After entering the SaaS offering, as an end user I want the SaaS offering to provide me the services which are appropriate for my role and learning context. This means it displays my name, offerings related to my role as teacher and/or learner, and a work environment prepared to connect to other end users participating in the same groups / courses. I want to identify group / course members based on their names.

1.3.4 Onboarding of new IDPs

As administrator of a school authority or federal state IDP, I want to have only one technical onboarding (configuration) with the ID Broker environment to implement the above listed end user use cases for all SaaS offerings / service providers. This onboarding should be as easy as possible while providing a secure and trustworthy connection. The needed steps are well documented.

As administrator I also want to have clear responsibilities in case a problem occurs and troubleshooting needs to be done.

As person responsible for the IDP and the information stored in it, I want to be sure that the handling of the data is done well (only minimal data is transferred, systems are secure) and the legal framework has been clarified (i.a. a data processing contract is signed).

1.3.5 Onboarding of new Service Providers

As administrator of a service provider, I want to have only one technical onboarding (configuration) with the ID Broker environment to implement the above listed end user use cases for all IDPs (school authorities and federal states). This onboarding should be as easy as possible while providing a secure and trustworthy connection. The needed steps are well documented.

As administrator I also want to have clear responsibilities in case a problem occurs and troubleshooting needs to be done.

As person responsible for the provided Service and the information stored in it, I want to be sure that the handling of the data is done well (only minimal data is transferred, systems are secure) and the legal framework has been clarified (i.a. a data processing contract is signed).

1.3.6 Operation if the ID Broker environment

As operator of the ID Broker environment I want to know how to install the environment, which services it has to provide and where to find information about the services, their architecture / modules, KPIs about the health state of the services and information where to find log messages. I expect to have a way to do a fully automated setup.

1.3.7 Univention as software vendor

As software vendor I want to maintain a solution which has as much overlap to my existing and established software stack as reasonable for the given use cases. I want to have the same development process as for other modules, including build and test procedures.

1.4 Requirements and demarcation

1.4.1 Requirements

Functional requirements

- Single sign-on (SSO) for end users while accessing service providers with school authority as leading identity provider.
- Information retrieval for service providers about the group memberships of an authenticated current user: granting access to the information is based on the users SSO session, so only information about a currently authenticated user can be retrieved.
- The unique identifier of an users has to be an individual pseudonym for each service provider: in case of a data breach of a service provider, there must not be any individual identifier of an user that allows to make a connection to the users data at any other service provider. It might be needed to extend this pseudonymization also to group identifiers.
- For security reasons, user authentication / “session” don’t last more than 6 hours. Afterwards the IDP of the school authority needs to be involved and might extend the session without asking the end user for the password.
- Provisioning of user and group information is limited to the scope of the school authority which authenticated against the Provisioning API.
- Any data retrieval API (initially the “Self-disclosure API”) limits access to data to the scope of the authenticated user: To access the API an authentication as end user is needed, the data to retrieve is data about the user and his or her context (i.e. learning groups). Detailed requirements will be added in the individual chapters.
- Adding school authorities is done in one configuration step for the school authority and the ID Broker operator and provides the school authorities users access to all current and future service providers.

- Adding service providers is done in one configuration step for the service provider and the ID Broker operator and provides access to the service for users of all current and future school authorities.
- In a future version of the document a service to de-pseudonymize a user will be introduced.

Nonfunctional requirements

- The solution has to follow European laws, this includes but is not limited to:
  - Data processing agreements have to be concluded with both service providers and school authorities. This is not part of the technical implementation but will be done as part of the onboarding process.
  - All data processing needs to be done under European jurisdiction (i.e. contracted operators and service providers need to be located in the EU).
  - Data storage is limited to the absolute needs for operation and functionality.
- UCS versions
  - School authority deployments need to support initially UCS 4.4 and in 2022 UCS 5.0.
  - ID Broker deployment shall be based on UCS 5.0 (to avoid a later migration from UCS 4.4 to UCS 5.0).
- Leading source of information is the school authority.
- Number of named users is expected to be about 100.000 (one hundred thousand) initially and 1.000.000 (one million) by the end of 2022.
- For all end user use cases the ID Broker has to ensure suitable response times and availability:
  - Relevant for these use cases are single sign-on and user information retrieval.
  - Suitable response times are expected as <0.5 seconds in >90% of all requests under peak load.
  - Peak load is initially expected to be 10% of named users per hour. This is:
    - Initially: 150 end user logons and information retrieval requests per minute.
    - By the end of 2022: 2000 end user logons and information retrieval requests per minute.
  - Availability of 99.99% of “learning hours”: Monday to Sunday 5:00 - 23:00.
- For provisioning use cases availability and processing requests are lower:
  - Outages of less than 5 Minutes can occur at any time
  - Peak loads to be handled are:
    - Initial provisioning of a large school authority: 500.000 new identities and corresponding groups in 5 days.
    - Change of all named users and corresponding groups in 6 weeks (summer holidays).
- Processed data has to be covered by contracts following EU data protection regulations (*Vertrag zur Auftragsdatenverarbeitung*).

1.4.2 Demarcation

The ID Broker must not:

- introduce a new account and/or new authorization information (new password) for users.
- give full access to stored data to any service provider (access is only allowed in the context of an authenticated end user).
- store any information not needed to process the defined use cases. Data not to be stored includes but is not limited to: passwords, contact information or any personal data, long term logs or any data that might be used for movement profiles not needed for fault analysis)

The ID Broker should not:

- be visible to the end users - the ID Broker mediates, but is only visible to administrators. Exceptions might occur in case of error handling.

1.5 Stakeholder

Stakeholders who have interest in the ID Broker and whose interests should be taken into account:

- School authorities
- End users (learners / students, lecturers / teachers, parents / legal guardians)
- Service providers
- Univention software development
- Univention operations

CHAPTER

TWO

HIGH LEVEL ARCHITECTURAL OVERVIEW

Let's have a high level look at the architecture. We have two diagrams explaining the same elements, and both diagrams are simplified. First we have a look at the participants, and after that we will learn about the action that is taking place.

2.1 Participants

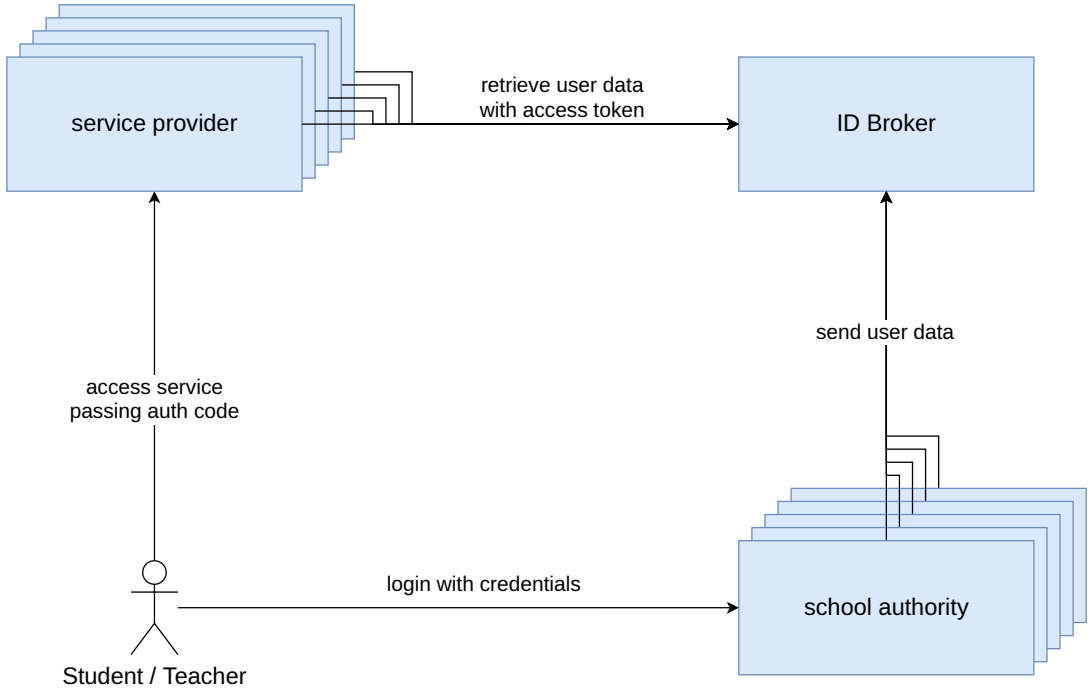


Fig. 2.1: Overview ID Broker

Student / Teacher
: The person that wants to login and access resources the *service provider* offers. To customize the resources, the service provider requires the users name, school and group membership.

School authority
: The entity that manages student data. The school authority also has an IDP to authenticate students, and a school portal for the login link that the student uses. A limited amount of user data is sent to the ID Broker.

Service provider
: The service that contains learning resources, but no student data (for example Bettermarks). Login requests are redirected to the IDP of the *school authority* the user belongs to.

ID Broker
: The service that brokers student data and login processes. This allows integrating multiple learning resources

and making them available to students of multiple school providers, without the *service providers* and *school authorities* having to communicate with each other. *Service providers* can retrieve metadata from the ID Broker about currently logged in users.

2.2 Components

Now let's uncover a few details.

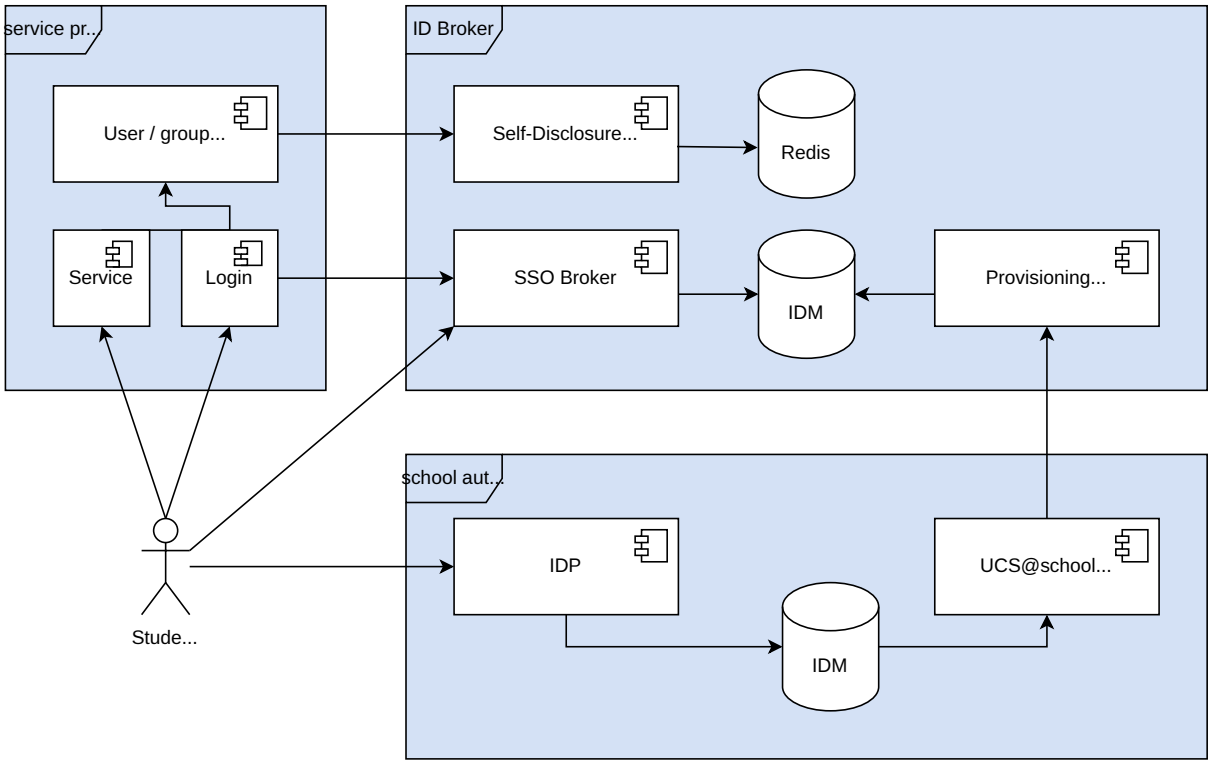


Fig. 2.2: ID Broker components

UCS@school ID Connector
The *UCS@school ID Connector* sends selected user data from the school authorities IDM to the *Provisioning API* on the ID Broker.

Provisioning API
The *Provisioning API* receives user data from school authorities and stores it in the ID Brokers IDM in a multi-tenant safe way.

UCS@school IDP
The *IDP* of the school authority is the only one to ever see the users credentials. Authenticated users receive a ticket that they send to the *SSO Broker*.

SSO Broker
The *SSO Broker* can validate the school authorities ticket and give the user a ticket to access a resource of the *service provider*.

Service Provider
The user send the *service provider* this ticket, which it uses to retrieve data about the connected user from the *Self-disclosure API* on the ID Broker.

Self-disclosure API
The *Self-disclosure API* provides the *service provider* with data about the connected user. The *Provisioning API* had stored that data earlier in the ID Brokers IDM.

Redis

The *Redis* cache is used by the Self-disclosure API to increase it's performance while accessing data about the connected user.

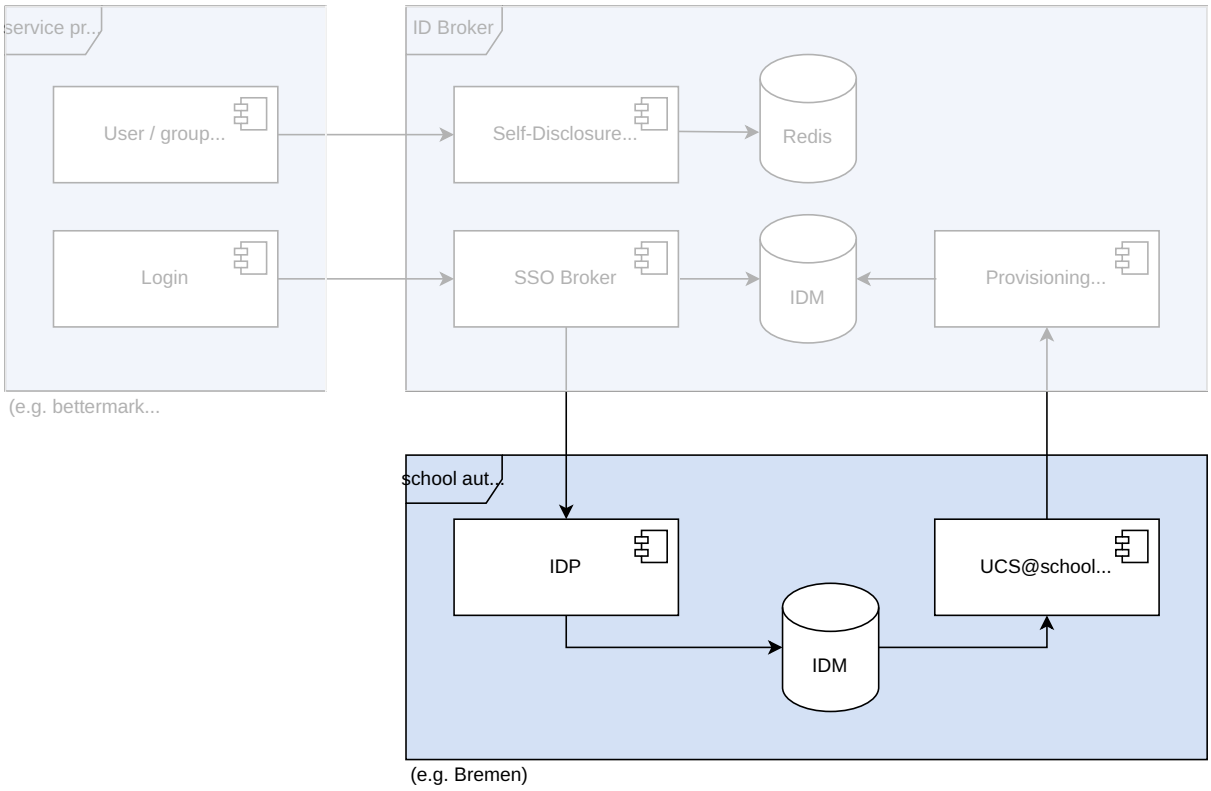
CHAPTER

THREE

SCHOOL AUTHORITY COMPONENTS

In this chapter the components that communicate with the ID Broker system are described.

Service providers use the ID Broker for authentication and to retrieve information about logged in users. School authorities do the actual authentication and send their users data to the ID Broker. The ID Broker system provides interfaces for multi tenant authentication, user data storage and retrieval.



3.1 School authorities / schools

Analogous to the ID broker system, the UCS@school platform also forms the basis at the school authority, on which various sub-components implement the required interfaces. The use of the UCS@school platform is a mandatory requirement.

The following UCS@school sub-components are relevant for the communication with the ID broker:

3.1.1 Identity management

Together with other UCS core components such as UMC and UDM, OpenLDAP forms the identity management (IDM) at the school authority. All relevant school objects, such as schools, users and group memberships, are administered via the IDM.

3.1.2 Identity provider

The identity provider (IDP) is also a module of UCS@school, which is responsible for the authentication of users. For this purpose, multiple authentication mechanisms such as OpenID Connect or SAML are supported by the IDP. The IDP can usually be accessed from the outside in order to connect external services to the UCS@school domain of the school board. In this scenario, the ID broker assumes the role of an external service, and makes appropriate authentication requests to the IDP. In order to answer the authentication requests, the IDP accesses the local user data of the IDM.

3.1.3 UCS@school ID Connector

For the provisioning of the ID broker, another component is required, which is also part of the UCS@school platform. The UCS@school ID Connector offers the possibility to connect a UCS@school domain to another UCS@school domain (here the ID Broker system) and to provision it with user data.

To use the UCS@school ID Connector in conjunction with the ID Broker, the *ID Connector Plugin* is required.

- A management API is accessible at `https://FQDN/ucsschool-id-connector/api/v1/docs`.
- The API should **not** be made accessible to the public, as it is only used for configuration purposes.
- The official documentation: https://docs.software-univention.de/ucsschool-id-connector/
- The source code is available at https://git.knut.univention.de/univention/components/ucsschool-id-connector

ID Connector Plugin

This plugin for the *UCS@school ID Connector* is triggered by changes in the school authorities IDM (LDAP), i.e. creation, modification and deletion, of all UCS@school users and school groups which are configured to be connected to the ID Broker.

Note: With version `1.3.18` of the package `id-broker-id-connector-plugin` the tool **`manage_schools_to_sync.py`** can be used to or and remove schools from the ID Broker. The new default is be that new schools have to be added manually. The old behavior, i.e. all current and future schools are synchronized, still works after the upgrade.

If a change is detected, the plugin uses the *Provisioning API* to modify the user data on the ID Broker. If an object is part of a school, which is not yet existing on the ID Broker, this school is created automatically on the ID Broker.

The plugin for the UCS@school ID Connector sends the following data from the school authority to the Provisioning API:

- **Users**: Only UCS@school users are sent, "normal" users are ignored. The attributes sent are: `entryUUID` (a unique object ID in the IDM of the school authority), `firstname`, `lastname`, `username` and `context`. Where `context` is a structure that contains the names of the schools the user is a member of, the groups in those schools and the users role (student, teacher, staff) in them.
- **School classes** and **Workgroups**: Only the school groups of users that should by synchronized are sent. The attributes sent are: `name`, `description` (display name), `school` and `members`.
- **Schools**: Only the schools of users that should by synchronized are sent. The attributes sent are: `name` and `displayName`.

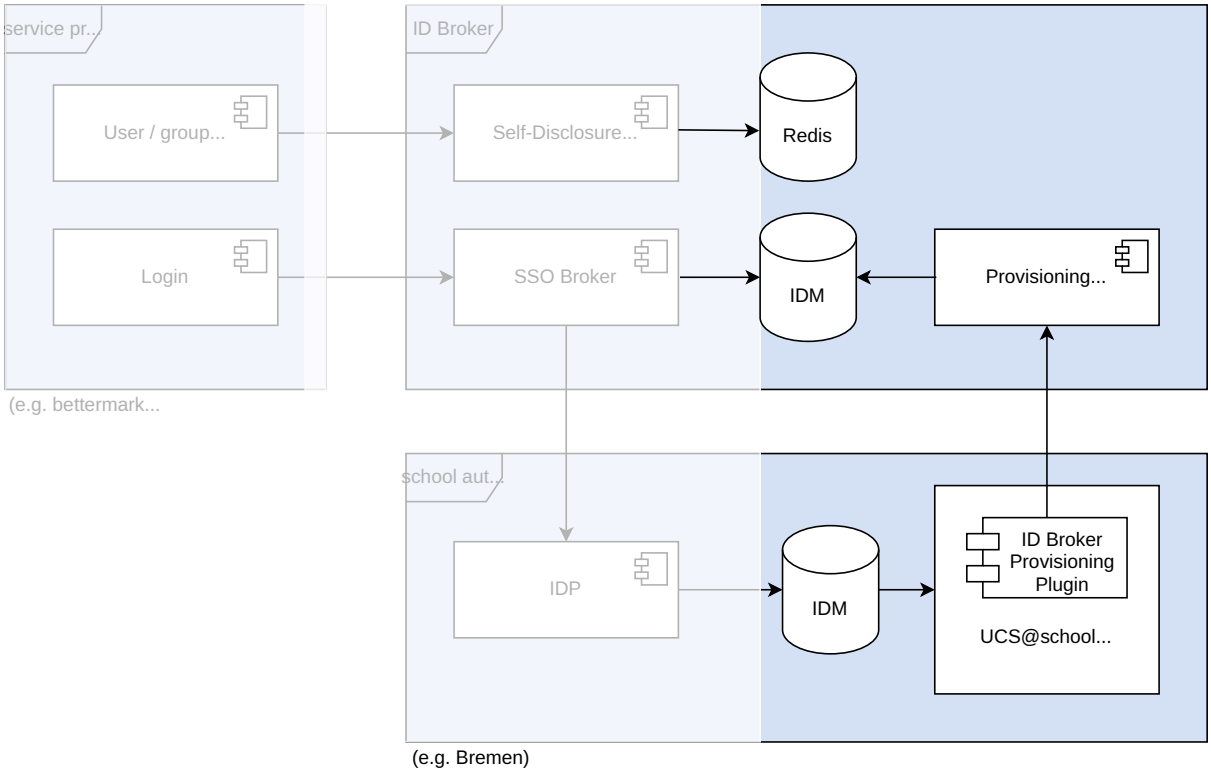


Fig. 3.1: UCS@school ID Connector and Provisioning API plugin

- The documentation for school authorities is available at https://docs.software-univention.de/idbroker-school-authority-manual/index.html
- The source code is available at https://git.knut.univention.de/univention/components/ucsschool-id-connector/-/tree/master/src/plugins

CHAPTER

FOUR

ID BROKER COMPONENTS

In this chapter the components that make up the ID Broker system are described.

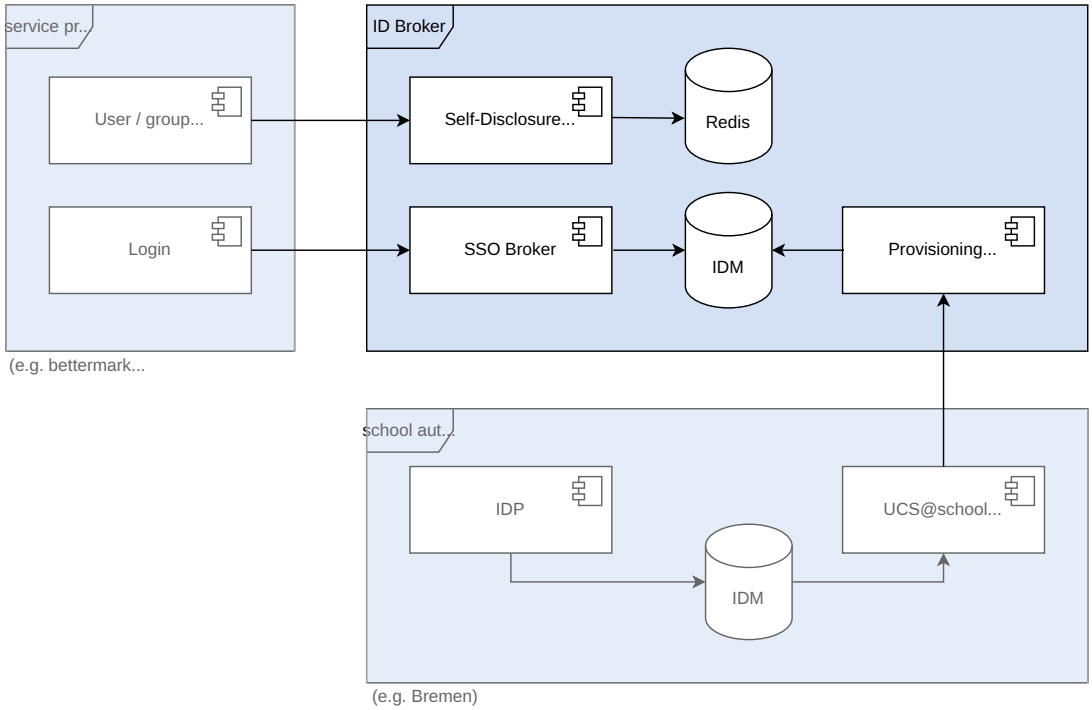


Fig. 4.1: Interaction of components

Service providers use the ID Broker for authentication and to retrieve information about logged in users. School authorities do the actual authentication and send their users data to the ID Broker. The ID Broker system provides interfaces for multi tenant authentication, user data storage and retrieval.

4.1 Modules

The base for an ID Broker system is the UCS@school platform, on top of which various components implement the required interfaces.

4.1.1 UCS / UCS@school core system

UCS@school components, like the UCS@school Kelvin REST API[1], are build on top of UCS' core components OpenLDAP, Univention Directory Manager (UDM)[2] and the UDM HTTP REST API[3].

Relevant for the ID Broker system are:

LDAP structure

Schools are represented as `OU` nodes with containers for users, groups, computers and so on below them.

All school object belong to a single OU, except users. User object are stored inside one of its schools OUs, but have an additional attribute which lists all schools (OUs) they are members of.

Usernames and group names must be unique. Under the hood, names of school groups are prefixed with the OUs name, so the same school groups name can be used by multiple schools.

A regular UCS@school system represents the domain of one school authority with all its schools, users, groups etc. For the multi tenant feature of the ID Broker, the names of objects that must have unique names in LDAP are internally prefixed with the identifier of the tenant or replaced with a UUID.

UDM

Univention Directory Manager (UDM)[4] is a Python library that adds business logic on top of LDAP objects. UDMs features can be used through its `Python UDM interface`[5], the UDM command line[6] or the UDM REST API[7].

The UDM extended attributes[8] feature is used to register additional LDAP attributes required for the ID Broker system. For example the new user attribute `brokerID` is used to map a UUID to the username of a tenants user. Another attribute will be used to map between service provider specific aliases and the real user account names. All LDAP attributes registered with UDM are accessible as UDM properties in the *UDM REST API*.

UDM REST API

UCS provides a the UDM REST API[9] which can be used to inspect, modify, create and delete UDM objects via HTTP requests. All UDM modules and their attributes are accessible through it. The UDM REST API converts the types of most attributes from their LDAP string representations to more useful JSON representations. It does not do that for extended attributes though.

- The UDM REST API is accessible at `https://FQDN/univention/udm/`.
- The UDM REST API should **not** be made accessible to the public, as it will only be accessed by the *Kelvin REST API*.
- The source code is accessible at https://git.knut.univention.de/univention/ucs/-/tree/5.0-1/management/univention-directory-manager-rest

[1] https://docs.software-univention.de/ucsschool-kelvin-rest-api/index.html
[2] https://docs.software-univention.de/developer-reference/5.0/en/udm/index.html#chap-udm
[3] https://docs.software-univention.de/developer-reference/5.0/en/udm/rest-api.html#udm-rest-api
[4] https://docs.software-univention.de/developer-reference/5.0/en/udm/index.html#chap-udm
[5] https://docs.software-univention.de/ucs-python-api/univention.udm.html#module-univention.udm
[6] https://docs.software-univention.de/manual/5.0/en/central-management-umc/udm-command.html#central-udm
[7] https://docs.software-univention.de/developer-reference/5.0/en/udm/rest-api.html#udm-rest-api
[8] https://docs.software-univention.de/developer-reference/5.0/en/udm/package-extended-attributes.html#udm-ea
[9] https://docs.software-univention.de/developer-reference/5.0/en/udm/rest-api.html#udm-rest-api

UCS@school Kelvin REST API

The UCS@school Kelvin REST API[10] provides HTTP endpoints to create and manage UCS@school domain objects like school users, school groups and schools (OUs). The Kelvin REST API internally uses the UCS@school library to add business logic on top of regular UDM user, group and computer objects. The result are for example complex user and server roles and finer grained authorization. To handle UCS@school objects, use the Kelvin REST API and not the UDM REST API, as it will take of data consistency. The Kelvin REST API uses the UDM REST API to communicate with the LDAP database and the Open Policy Agent[11] for authorization.

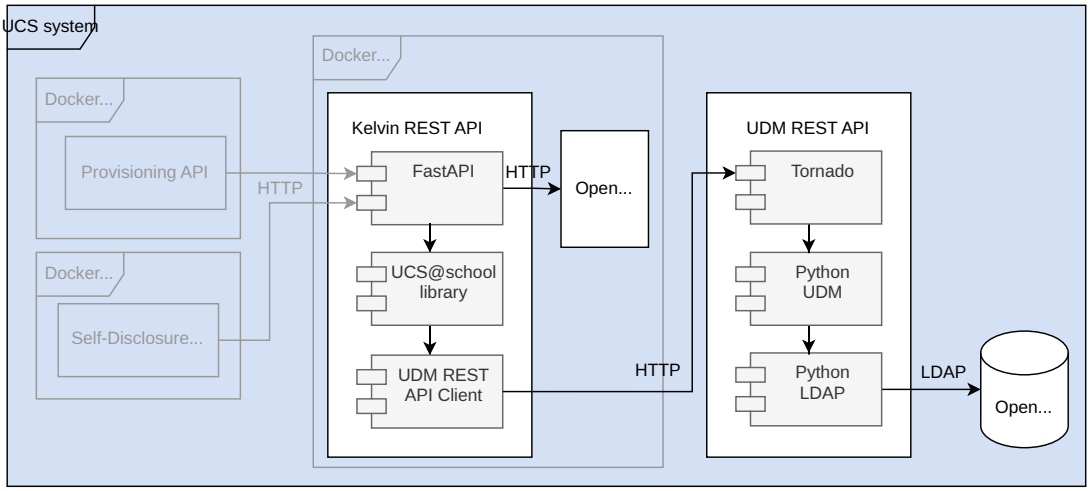


Fig. 4.2: Kelvin REST API components and connections

- The Kelvin REST API is accessible at `https://FQDN/ucsschool/kelvin/v1/docs`.
- The Kelvin REST API should **not** be made accessible to the public, as it will only be accessed by the *Provisioning API* and the *Self-disclosure builder*.
- The source code is accessible at https://git.knut.univention.de/univention/ucsschool/-/tree/feature/kelvin/kelvin-api

4.1.2 Provisioning API

Users and groups have to be created in the ID Broker system. Those users originate from the school authority systems. The *Provisioning API* is a REST API with methods and routes to read, create, update and delete users, school groups and schools.

The *UCS@school ID Connector* of each school authority uses the Provisioning API to send user and group data to the ID Broker system. Each school authority has an account in the ID Broker that allows it modify only its own objects. All objects managed through such an identity share a common namespace implemented as prefixes for the username / group names / OU names.

The Provisioning API transparently adds prefixes when talking to internal systems and removes them when talking to external ones. It acts like an *adapter* between the UCS@school ID Connector and the Kelvin REST API.

The Provisioning API is responsible for generating service provider specific pseudonyms. Separate pseudonyms are generated for each service provider and stored in separate attributes in the users/groups/OUs LDAP objects. A mapping from service provider ID to LDAP attribute name is retrieved from LDAP. Additionally a mapping from service provider ID to a secret password (used as *salt* in the generation of the pseudonym) is retrieved from LDAP. Each pseudonym is generated as a *hash* from the following three values:

- `entryUUID` of the object in the school authorities LDAP (assumed to be a globally unique string)
- service provider specific secret (the *salt*, known only to the ID Broker system)

[10] https://docs.software-univention.de/ucsschool-kelvin-rest-api/index.html
[11] https://www.openpolicyagent.org/

- school authority ID

The service provider specific secret prevents cooperating service providers to identify common users.

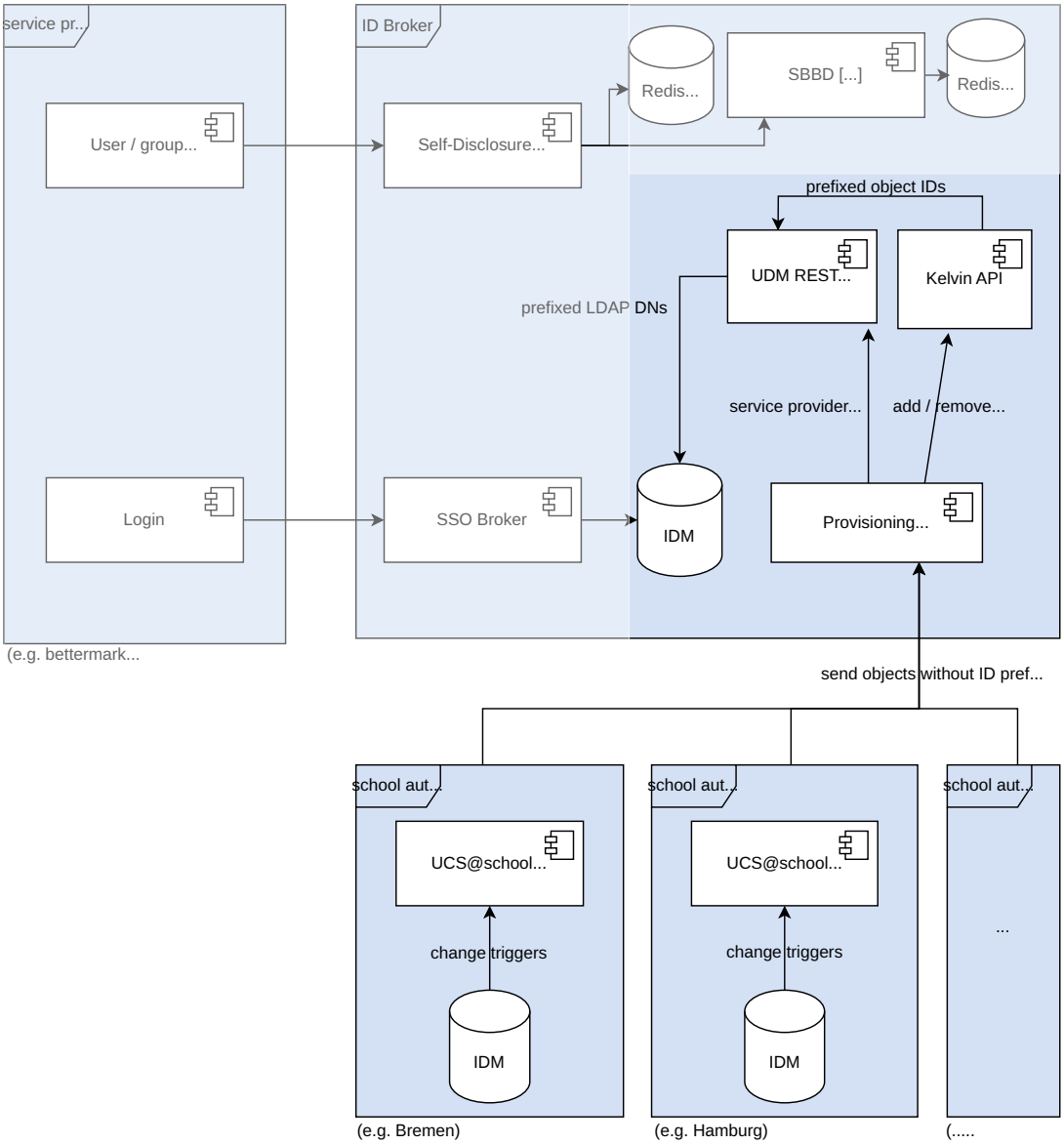


Fig. 4.3: Provisioning API communication

- The API is accessible at `https://FQDN/ucsschool/apis/docs`.
- The source code is accessible at https://git.knut.univention.de/univention/components/ucsschool-api-plugins/id-broker-plugin/-/tree/main/provisioning_plugin

4.1.3 Self-disclosure API

The design goal of the *Self-disclosure API* is to receive and send only service provider specific pseudonyms instead of clear text user IDs and other personal information. To make the services usable, the clear text values of some fields in the Self-disclosure API are transmitted (cf. section *Future evolutions of the pseudonymization* (page 22)). Fig. 4.4 shows the API communication of the self disclosure API as described in the following paragraphs.

The *Self-disclosure API* is one example of an HTTP API where a content provider can fetch user data customized to their needs. It is implemented as a plugin for the *UCS@school APIs* app. It runs in a Docker container on an UCS@school system.

The *Self-disclosure API* uses the Redis[12] database populated by the *Self-disclosure database builder* (page 20) to fetch user and group data.

The client of the the API, e.g. *Bettermarks*, needs an auth code to get access to the API. This token is typically passed on from a student's or teacher's browser. The student or teacher in return has received this auth code from the IDP of its school authority.

The ID in the tokens subject field is the pseudonym of the requesting user. The ID in the resource request parameter (in the URL) is the pseudonym of the user or group that information is requested about.

Separate pseudonyms are generated for each service provider and stored in separate attributes in the users/groups/OUs LDAP objects. A mapping from service provider ID to LDAP attribute name exists in LDAP. The Self-disclosure API retrieves that mapping for the connecting service provider using the Redis cache.

When the Self-disclosure API has to *lookup* an object, it searches for the supplied pseudonym in the Redis cache.

When the Self-disclosure API has to *return* an object, instead of sending the user ID, it sends the service provider specific pseudonym of that object.

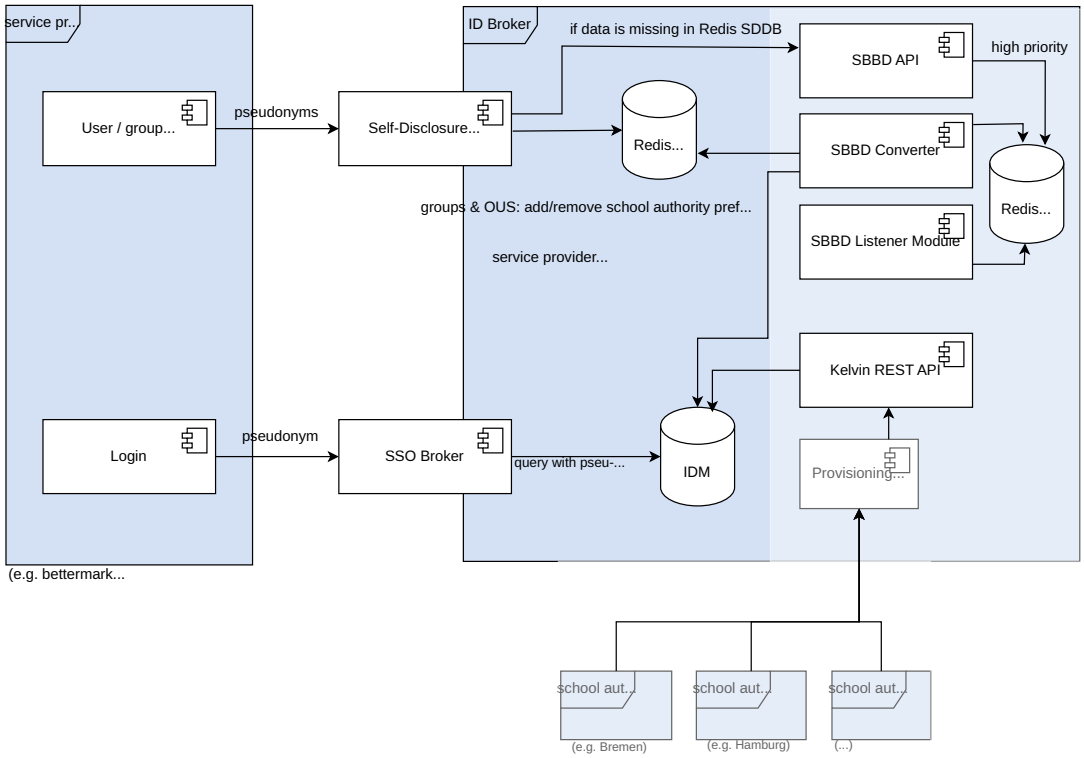


Fig. 4.4: Self-disclosure API communication

- The API is accessible at `https://FQDN/ucsschool/apis/docs`.
- The source code is accessible at https://git.knut.univention.de/univention/components/ucsschool-api-plugins/id-broker-plugin/-/tree/main/self_disclosure_plugin

[12] https://redis.io/

4.1.4 Self-disclosure database builder

The design goal of the Self-disclosure database builder is to improve the performance of the *Self-disclosure API* (page 19). It uses a Redis database to build a cache of user, group, and service provider mappings stored in LDAP.

The diagram below shows a detailed view of the components involved. Fig. 4.5 shows a detailed view of the components involved.

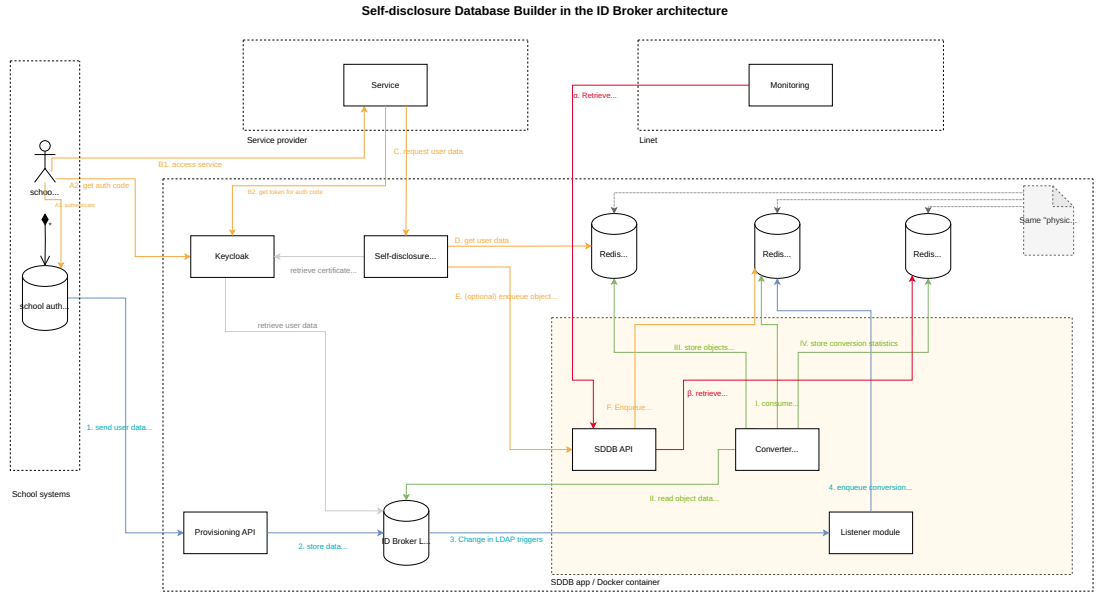


Fig. 4.5: Self-disclosure Database Builder

Changes pushed to the ID Broker LDAP trigger a listener module (3) which enqueues the insert/update or delete event to a Redis database table which acts as a queue (4). The converter daemon consumes these events (I), reads from Kelvin and LDAP (II) and saves the data in a Redis table which saves the complete object (III).

The *Self-disclosure API* (page 19) uses the data stored in the SDDB database to get user and group data (D). If the data hasn't yet been inserted by the converter daemon because the replication isn't yet finished, the Self-disclosure API query the SDDB API to add the object to the high priority queue so it will be there when the object is requested again (F).

During this process statistics are written by the internal components of the SDDB builder in a third table (IV). They can be requested through a prometheus[13] endpoint of the SDDB API. The endpoint does not require authentication since it does not offer any private information.

- The API is accessible at `http://FQDN/ucsschool/id-broker-sddb-builder/v1/docs`.
- The source code is accessible at https://git.knut.univention.de/univention/ucsschool-components/id-broker-self-disclosure-db-builder

4.1.5 SSO Broker

The main job of the SSO Broker component is to handle multiple-tenant authentication, using pseudonyms. This involves the student (or her browser) doing the login and passing authentication tokens/tickets back and forth.

The *SSO Broker* participates in the following communications:

- The *student* gets sent to the SSO Broker upon first login (a redirect from the school portal). This first step is part of an OIDC flow. The SSO Broker then sends the student to the *school authority's IDP*, to do SAML authentication there. This is done using a real user identifier. The student returns to the SSO Broker with her SAML ticket.

[13] https://prometheus.io/

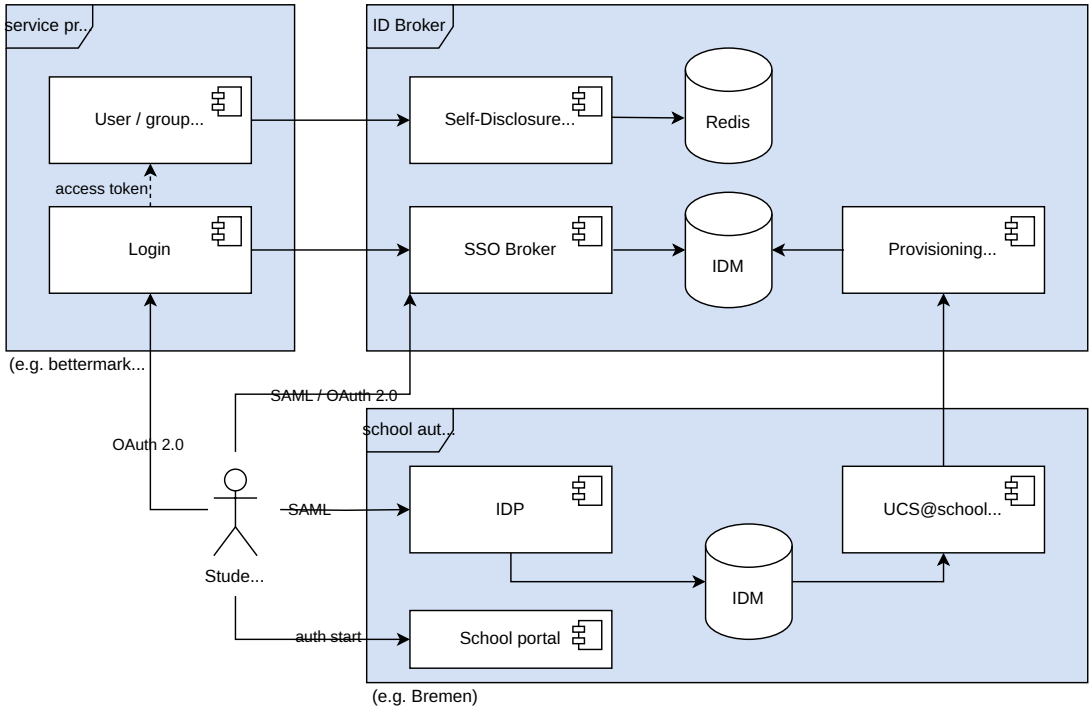


Fig. 4.6: SSO Broker communications

- The SSO Broker then needs to get a *service provider* specific pseudonym. This information is provided by the ID Broker IDM system, which also contains other user data provided by the school authority. An auth code valid for the (service provider specific) pseudonym is then given to the student, who passes it on to the service provider.
- The service provider then swaps the auth code for both an access token and an ID token at the SSO Broker. The ID token (containing the pseudonym) is consumed by the service provider, while the access token can be used to request more data about the student (referred to by the pseudonym) at the *Self-disclosure API* (page 19).

The SSO Broker is implemented using Keycloak.

The SSO Broker is available at:

- for OIDC at `https://`*`FQDN`*`/auth/realms/`*`SERVICE PROVIDER ID`*`/protocol/openid-connect`
- for SAML at `https://`*`FQDN`*`/auth/realms/`*`SERVICE PROVIDER ID`*`/broker/saml`

Information about the configuration of Keycloak can be found at https://univention.gitpages.knut.univention.de/id-broker/operations-manual/

- See chapter *ID Broker architecture and flows* (page 29) for an in-depth explanation of the authentication mechanisms.
- See chapter keycloak-overview for a description of the Keycloak setup.

4.2 Pseudonymization

A core concept of the ID Broker is the pseudonymization of user data towards the service providers. It is not only desired to hide the clear text values for names etc. from service providers but also prevent data analysis between multiple providers. To that effect each user, group and school OU in the ID Broker system get's a separate pseudonym for each service provider which is saved in its own LDAP attribute (*idBrokerPseudonym0001* through *idBrokerPseudonym0030*).

4.2.1 Management of Service Providers

To enable each component of the ID Broker to always have access to the correct pseudonyms for each service provider the pseudonyms will be saved as individual LDAP fields on users, groups and school OUs. Those fields are indexed to ensure quick searches. To know which field belongs to which service provider a mapping from provider name to LDAP field name has to be created and maintained as well. Since this mapping has to be available on the host and its docker containers alike, saving this mapping in the LDAP is the most obvious solution.

To manage service providers in the ID Broker, the script *manage-service-providers* can be used. It provides functionalities to add, delete and show the mappings as well as the secrets. When a new service provider is added, all existing users, groups and school OUs receive the corresponding pseudonym.

The steps which are needed to configure Keycloak are described in Backup - SSO Service - Keycloak[14].

4.2.2 Form of the Pseudonyms

The primary identifier of any group, user or school OU object in the ID Broker system is its `entryUUID` from the school authority, where it is originating from. To ensure that an objects pseudonyms are recoverable in an event of data loss or sync errors, it should be derived from said `entryUUID`. Thus the pseudonym is generated as

```
pseudonym_service_provider1 = blake2b(salt=service_provider1_salt, person=school_
↪authority, data=entryUUID)
```

where *blake2b* is a hashing algorithm which returns a string in the ASCII space with no more than 128 symbols and *service_provider1_salt* is a previously generated secret string which is unique to each service provider.

4.2.3 Generation of pseudonyms

The generation of pseudonyms happens primarily during user, group or school OU creation in the Provisioning API. The system will automatically create pseudonyms for all known service providers at that time. When a new service provider is added to the ID Broker, it is necessary to execute the script that generates pseudonyms for the new service provider for all users and groups that already exist on the system (*manage-service-providers*, see above).

4.2.4 Future evolutions of the pseudonymization

The pseudonymization in its current form states that every user of every connected school authority gets a pseudonym for every existing service provider and thus is usable with it. Future iterations could implement the following ideas and features:

- Service providers can be activated for users and groups on a school authority level
- Service providers can be activated for users and groups individually (filtered by school, school_class, etc) by the school authority
- The script for generating pseudonyms for new service providers is transformed into a small service which can react to new service providers and generates pseudonyms in an intelligent and load balanced way.

[14] https://univention.gitpages.knut.univention.de/id-broker/operations-manual/installation.html#install-id-broker-backup-sso-service

To make the services usable the clear text values of some fields in the `id token` as well as the Self-disclosure API are transmitted. This renders the current pseudonymization ineffective. This is known to all parties and will be removed in the next project phase as soon as the de-pseudonymization component is implemented.

4.3 Scaling

The APIs that are accessible from outside the ID Broker system are the *Provisioning API* and the *Self-disclosure API*. The Provisioning API uses the *Kelvin REST API* to access user / group data, which in turn uses the *UDM REST API* to access the underlying database. The Self-disclosure API uses the Redis cache build by the Self-disclosure database builder.

As the Provisioning API and the Self-disclosure API have very different requirements regarding availability and response time, using separate systems is recommended.

In previous tests, *with preliminary ID Broker system components*, the UDM REST API was the bottleneck. Depending on the hardware its response times are limited by I/O or CPU time.

The current design is to keep the components of each of the Provisioning API chain ("Provisioning -> Kelvin -> UDM") on the same host and to not do any load balancing between the internal components. The Self-disclosure API and the Self-disclosure database builder are installed on separate systems.

Vertical scaling can be done through higher CPU core count and faster disk I/O. To a degree also with more memory for caching. An optimal distribution of CPU cores to the worker processes of the three REST APIs has not yet been explored and may vary depending on the hardware.

Horizontal scaling can be done by load balancers in front of those systems. Load balancers can distribute the load depending on the response time of the front-end APIs. Care must be taken with regards to tokens handed out by front-end APIs. Either sticky HTTP sessions are required or shared keys on the servers for token verification. This is probably only relevant for the Provisioning API, as the Self-disclosure API will not hand out tokens.

API clients must be implemented with fault tolerance regarding lost sessions, as load balancers may have to move their connection when a server is down / being updated.

CHAPTER

FIVE

INTERACTIONS BETWEEN COMPONENTS

The ID Broker architecture has been designed in a way that allows the users of multiple school authorities to securely access the resources of multiple service providers, without the school authorities and service providers having to communicate with each other. The users password is only send once to the IDP of its school authority. The service providers do not have to store any user information.

5.1 Authentication and user data retrieval

When a user wants to access a resource of one of the service providers, she needs to authenticate herself and the service provider requires some data about the user to provide an individualized service.

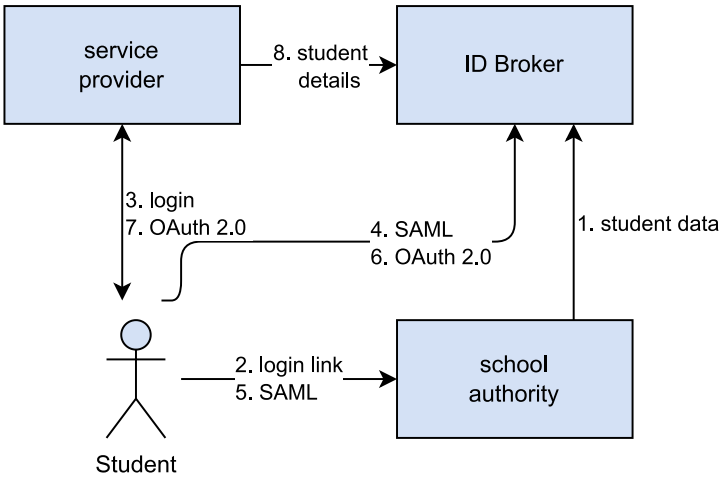


Fig. 5.1: ID Broker - connections

The service provider however does not do the authentication itself. It does not even know the name of the school authority the user belongs to or the address of its IDP. So the service provider redirects the user to the ID Broker which in turn redirects the user to the IDP of its school authority.

The ID Broker can verify the signature of the school authorities IDP and give the user a ticket. The user passes that ticket to the service provider, which can now retrieve data about the user from the ID Broker.

1. student data
The school authority syncs student data to the ID Broker.

2. request service provider login at school authorities portal page
The student clicks a link on the school authorities portal page, and gets redirected to the service provider. This redirection is necessary to make the combination of SAML and OpenID Connect possible - the ID Broker needs to know which SAML backend should be used.

3. request login
The student requests a login at the service providers page, and gets redirected to the ID Broker.

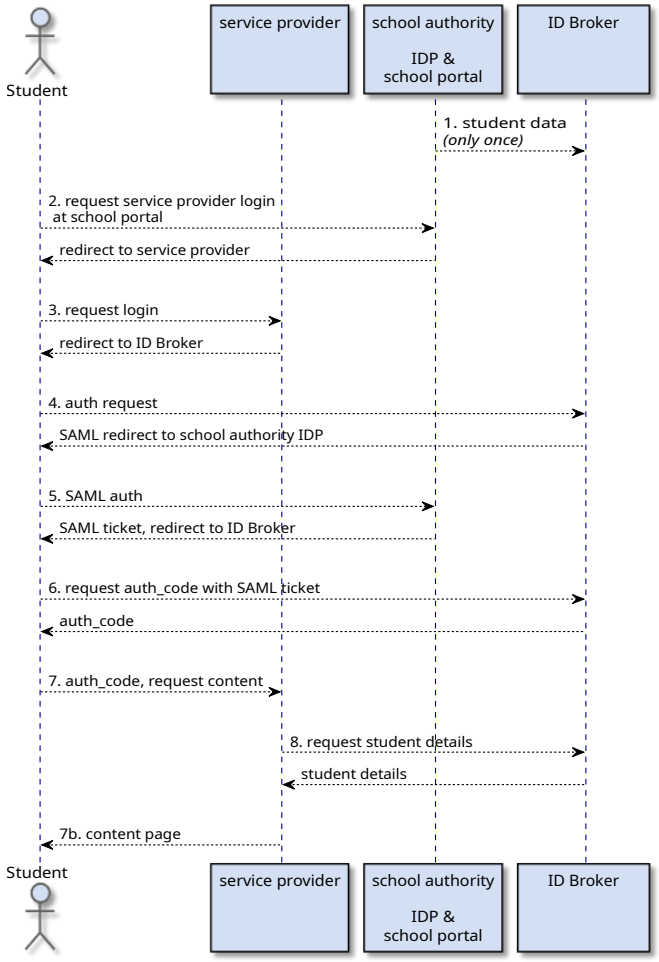


Fig. 5.2: ID Broker sequence: authentication and user data retrieval sequence *(simplified visualization)*

4. auth request
The ID Broker does not do the actual login, instead the student gets redirected to the school authority, which has an IDP (SAML) provider.

5. SAML auth
Here, the actual SAML auth happens, the student gets a SAML ticket, and is redirected to the ID Broker.

6. request auth_code
Using the SAML ticket, the user requests an auth_code from the ID Broker. The user gets redirected to the service provider.

7. auth_code, request content
The user passes on the auth_code while asking for the actual content page(s). The auth_code is exchanged by the service provider for an access token (this step is left out of the diagram for clarity reasons, but takes place in reality).

8. request student details
Using the access token, the service provider can now request user data from the ID Broker.

7b. content page
This is the continuation of step 7 - the student gets the requested content from the service provider.

CHAPTER

SIX

APPENDIX

6.1 ID Broker architecture and flows

This document is the result of researching different possibilities on how to implement SSO between the student and teachers, service providers and the school authorities.

Warning: Because of the research setup, `http://` is used everywhere, and not `https://`, which MUST be used in a real setup.

Warning: There are also other errors in the details of the recorded flow, which are going to be corrected. WIP.

6.1.1 Theory

One should be somewhat familiar with id_broker_architecture/saml and id_broker_architecture/oauth2 flows before reading the details of the ID Broker auth architecture. Also id_broker_architecture/understanding_jwt helps.

6.1.2 Requirements for the auth flow

One requirement is that the school authorities can keep on using SAML. We found that we can't just start with SAML and then 'hand over' to OAuth2 (see below *Auth code flow II - First SAML* (page 44)). Instead, we need to interweave OAuth2 and SAML flows.

6.1.3 ID Broker Flow

This presents the mix of SAML and OAuth2.

Mapping of terms and roles

- **student** - the student/browser using the application.
- **School portal** - the portal where the student clicks on a link to the *Service Provider (SP)* (e.g. *Bettermarks*).
- **School IDP (SAML)** - the *Identity Provider (IDP)* that speaks SAML and is provided by UCS.
- **SSO Broker (Keycloak)** - the SSO Broker runs APIs, and uses Keycloak to manage authentication.
- **Self-disclosure API** - an API that provides useful information to the service provider (e.g. *Bettermarks*).
- **service provider** - the actual application that wants to consume data.

Flowchart - "OIDC first"

The following figure shows the flow from OIDC, to SAML, to OIDC.

Details with messages

1. visit site

The initiate step is always the user/student visiting her school portal. We need this in order to get information about who the IDP is.

2. link: service provider

On the school portal she will find a link that sends her to the service provider, or the `testapp2.py` in our case. Please note the `idp_hint` - this is needed to inform Keycloak about which IDP to use as the backend.

```
http://10.205.2.110:5000/?idp_hint=ExampleSA2
```

- This writes the *idp_hint* to the session with the client

3. Request to protected resource on client

She follows the link to the protected resource.

```
http://10.205.2.110:5000/private
```

4. redirect:SSO Broker

The protected resource doesn't know her, so an OIDC flow gets initiated. This would be sent to the authorization server in OAuth2, and for us it is the SSO Broker. `Scope` describes what the *testapp* is asking authorization for:

```
Location: https://login.keycloak.idbroker.intranet/auth/realms/ID-Broker/protocol/
 ↪openid-connect/auth?
 client_id=python-client&
 redirect_uri=http%3A%2F%2F10.205.2.110%3A5000%2Foidc_callback&
 scope=openid+email+profile&
 access_type=offline&
 response_type=code&
 kc_idp_hint=ExampleSA2&
␣
 ↪state=eyJjc3JmX3Rva2VuIjogInJrT2dDb3ZUR25scElTR055eGtKdC1RT2tFYUY2dUhRIiwgImRlc3RpbmF0aW9uIjogIn
 ↪%3D%3D
```

- `kc_idp_hint` is based on `idp_hint` value in session

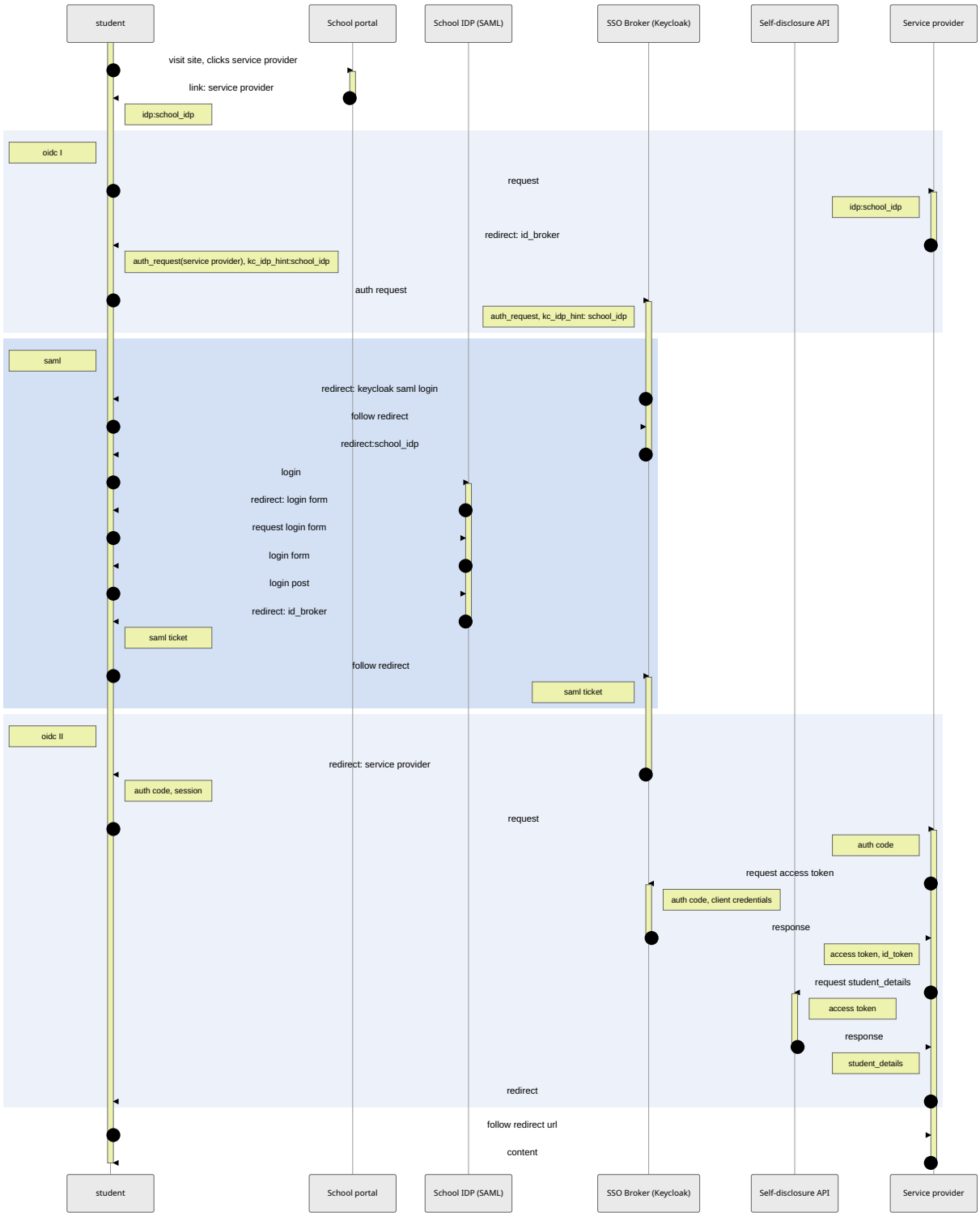


Fig. 6.1: OAuth2 / SAML / OAuth2 flow
click to get a larger version of the diagram

5. auth request

She follows the redirect:

```
https://login.keycloak.idbroker.intranet/auth/realms/ID-Broker/protocol/openid-
↪connect/auth?
 client_id=python-client&
 redirect_uri=http%3A%2F%2F10.205.2.110%3A5000%2Foidc_callback&
 scope=openid+email+profile&
 access_type=offline&
 response_type=code&
 kc_idp_hint=ExampleSA2&
␣
↪state=eyJjc3JmX3Rva2VuIjogInJrT2dDb3ZUR25scElTR055eGtKdC1RT2tFYUY2dUhRIiwgImRlc3RpbmF0aW9uIjogIn
↪%3D%3D
```

The state contains:

```
{
    "csrf_token": "rkOgCovTGnlpISGNyxkJt-QOkEaF6uHQ",
    "destination": "eyJhbGciOiJIUzUxMiJ9.
↪Imh0dHA6Ly8xMC4yMDUuMi4xMTA6NTAwMC9wcml2YXRlIg.
↪fHOnX2dN17I7XYqRUbA8etZmpAfKe4yjPhlVrGDFfLBqbyI82iXgfB4k7FABYRmtWZy3uvdEhx51Utqe3Ek1wA
↪"
}
```

which means

```
{
  "destination": {
    "headers": {
      "alg": "HS512"
    },
    "payload": "http://10.205.2.110:5000/private",
    "signature":
↪"fHOnX2dN17I7XYqRUbA8etZmpAfKe4yjPhlVrGDFfLBqbyI82iXgfB4k7FABYRmtWZy3uvdEhx51Utqe3Ek1wA
↪"
  },
  "csrf_token": "rkOgCovTGnlpISGNyxkJt-QOkEaF6uHQ"
}
```

6. redirect: Keycloak SAML login

Keycloak doesn't hasn't got the browser authenticated yet. So the user gets a redirect to the SAML endpoint.

```
Location: https://login.keycloak.idbroker.intranet/auth/realms/ID-Broker/broker/
↪ExampleSA2/login?
 session_code=U7QqTnIs1CTL7ATh1cf_XFCaCi95oPE4v7Uwc5baH2s&
 client_id=python-client&
 tab_id=_NzbfJl6_b0
```

- set session cookies

7. follow redirect

The user follows to the SAML endpoint, asking for a login.

```
https://login.keycloak.idbroker.intranet/auth/realms/ID-Broker/broker/ExampleSA2/
↪login?
 session_code=U7QqTnIs1CTL7ATh1cf_XFCaCi95oPE4v7Uwc5baH2s&
 client_id=python-client&
 tab_id=_NzbfJl6_b0
```

8. redirect: School IDP

The endpoint doesn't know the user yet. But because of step (5) Keycloak knows to which IDP she needs to be sent. The request contains a proper SAML request.

```
Location: https://ucs-sso.school2.intranet/simplesamlphp/saml2/idp/SSOService.php
{
    "SAMLRequest": "< see below, base 64 encoded >",
    "RelayState": "4vuHEB1r-lkWNvElsxy4si8qgTCfnTM77J8Z7AIb5P8.qt_dqhNpMEY.python-
↪client"
}
```

Which means

```
<samlp:AuthnRequest xmlns:samlp="urn:oasis:names:tc:SAML:2.0:protocol"
                    xmlns="urn:oasis:names:tc:SAML:2.0:assertion"
                    xmlns:saml="urn:oasis:names:tc:SAML:2.0:assertion"
                    AssertionConsumerServiceURL="https://login.keycloak.idbroker.
↪intranet/auth/realms/ID-Broker/broker/ExampleSA2/endpoint"
                    Destination="https://ucs-sso.school2.intranet/simplesamlphp/
↪saml2/idp/SSOService.php"
                    ForceAuthn="false"
                    ID="ID_cc9da054-144b-4c42-8dbc-2008860f2f01"
                    IssueInstant="2022-02-01T12:29:59.874Z"
                    ProtocolBinding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-
↪Redirect"
                    Version="2.0">
    <saml:Issuer>https://login.keycloak.idbroker.intranet/auth/realms/ID-Broker/
↪broker/ExampleSA2/endpoint/descriptor</saml:Issuer>
    <dsig:Signature xmlns:dsig="http://www.w3.org/2000/09/xmldsig#">
        <dsig:SignedInfo>
            <dsig:CanonicalizationMethod Algorithm="http://www.w3.org/2001/10/xml-
↪exc-c14n#"/>
            <dsig:SignatureMethod Algorithm="http://www.w3.org/2001/04/xmldsig-more
↪#rsa-sha256"/>
            <dsig:Reference URI="#ID_cc9da054-144b-4c42-8dbc-2008860f2f01">
                <dsig:Transforms>
                    <dsig:Transform Algorithm="http://www.w3.org/2000/09/xmldsig
↪#enveloped-signature"/>
                    <dsig:Transform Algorithm="http://www.w3.org/2001/10/xml-exc-
↪c14n#"/>
                </dsig:Transforms>
                <dsig:DigestMethod Algorithm="http://www.w3.org/2001/04/xmlenc
↪#sha256"/>
                <dsig:DigestValue>P0P7K5x7yvgkllsr33iGeRtmLonN2R/T7oim2rb/vbw=</
↪dsig:DigestValue>
            </dsig:Reference>
        </dsig:SignedInfo>
        <dsig:SignatureValue>...</dsig:SignatureValue>
        <dsig:KeyInfo>
```

(continues on next page)

(continued from previous page)

```
            <dsig:X509Data>
                <dsig:X509Certificate>...</dsig:X509Certificate>
            </dsig:X509Data>
            <dsig:KeyValue>
                <dsig:RSAKeyValue>
                    <dsig:Modulus>...</dsig:Modulus>
                    <dsig:Exponent>AQAB</dsig:Exponent>
                </dsig:RSAKeyValue>
            </dsig:KeyValue>
        </dsig:KeyInfo>
    </dsig:Signature>
    <samlp:NameIDPolicy Format="urn:oasis:names:tc:SAML:2.0:nameid-format:transient
↪"/>
</samlp:AuthnRequest>
```

9. login

And of she goes to the UCS system, where the SAML auth is configured.

```
POST https://ucs-sso.school2.intranet/simplesamlphp/saml2/idp/SSOService.php
{
    "SAMLRequest": "< see above, base 64 encoded >",
    "RelayState": "4vuHEB1r-lkWNvElsxy4si8qgTCfnTM77J8Z7AIb5P8.qt_dqhNpMEY.python-
↪client"
}
```

10. Redirect: login form

Here she gets a redirect to the login form…

```
Location: https://ucs-sso.school2.intranet/simplesamlphp/module.php/core/
↪loginuserpass.php?
 AuthState=_008efe5d4af5d17b53125e25abd5cf49d80b6b4215%3Ahttps%3A%2F%2Fucs-sso.
↪school2.intranet%2Fsimplesamlphp%2Fsaml2%2Fidp%2FSSOService.php%3Fspentityid
↪%3Dhttps%253A%252F%252Flogin.keycloak.idbroker.intranet%252Fauth%252Frealms
↪%252FID-Broker%252Fbroker%252FExampleSA2%252Fendpoint%252Fdescriptor%26cookieTime
↪%3D1643718600%26RelayState%3D4vuHEB1r-lkWNvElsxy4si8qgTCfnTM77J8Z7AIb5P8.qt_
↪dqhNpMEY.python-client
```

11. Request login form

…which she requests…

```
Location: https://ucs-sso.school2.intranet/simplesamlphp/module.php/core/
↪loginuserpass.php?
 AuthState=_008efe5d4af5d17b53125e25abd5cf49d80b6b4215%3Ahttps%3A%2F%2Fucs-sso.
↪school2.intranet%2Fsimplesamlphp%2Fsaml2%2Fidp%2FSSOService.php%3Fspentityid
↪%3Dhttps%253A%252F%252Flogin.keycloak.idbroker.intranet%252Fauth%252Frealms
↪%252FID-Broker%252Fbroker%252FExampleSA2%252Fendpoint%252Fdescriptor%26cookieTime
↪%3D1643718600%26RelayState%3D4vuHEB1r-lkWNvElsxy4si8qgTCfnTM77J8Z7AIb5P8.qt_
↪dqhNpMEY.python-client
```

12. login form

And gets as a plain html form. She fills out the form…

13. login post

And posts the form to the UCS server.

```
POST https://ucs-sso.school2.intranet/simplesamlphp/module.php/core/loginuserpass.
↪php?
{
    "username": "84h5x0g7ex",
    "password": "univention",
    "AuthState": "_008efe5d4af5d17b53125e25abd5cf49d80b6b4215:https://ucs-sso.
↪school2.intranet/simplesamlphp/saml2/idp/SSOService.php?spentityid=https%3A%2F
↪%2Flogin.keycloak.idbroker.intranet%2Fauth%2Frealms%2FID-Broker%2Fbroker
↪%2FExampleSA2%2Fendpoint%2Fdescriptor&cookieTime=1643718600&RelayState=4vuHEB1r-
↪lkWNvElsxy4si8qgTCfnTM77J8Z7AIb5P8.qt_dqhNpMEY.python-client",
    "submit": ""
}
```

14. redirect SSO Broker

Upon successful login the user needs to do a POST to the Keycloak server. But we can't redirect to there, because http doesn't allow POST redirects. Hence, this is done in JavaScript.

15. follow redirect (POST)

So we have this POST to Keycloak with the SAML ticket in the body. We have left out the certificates for readability.

```
POST https://login.keycloak.idbroker.intranet/auth/realms/ID-Broker/broker/
↪ExampleSA2/endpoint
{
    "SAMLResponse": "< see below, base 64 encoded >",
    "RelayState": "4vuHEB1r-lkWNvElsxy4si8qgTCfnTM77J8Z7AIb5P8.qt_dqhNpMEY.python-
↪client"
}
```

```
<samlp:Response xmlns:samlp="urn:oasis:names:tc:SAML:2.0:protocol"
                xmlns:saml="urn:oasis:names:tc:SAML:2.0:assertion"
                ID="_93d030b688ccac198bc06adda4a76ff67eb8d18a25"
                Version="2.0"
                IssueInstant="2022-02-01T12:30:02Z"
                Destination="https://login.keycloak.idbroker.intranet/auth/realms/
↪ID-Broker/broker/ExampleSA2/endpoint"
                InResponseTo="ID_cc9da054-144b-4c42-8dbc-2008860f2f01">
    <saml:Issuer>https://ucs-sso.school2.intranet/simplesamlphp/saml2/idp/metadata.
↪php</saml:Issuer>
    <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
        <ds:SignedInfo>
            <ds:CanonicalizationMethod Algorithm="http://www.w3.org/2001/10/xml-
↪exc-c14n#"/>
            <ds:SignatureMethod Algorithm="http://www.w3.org/2001/04/xmldsig-more
↪#rsa-sha256"/>
            <ds:Reference URI="#_93d030b688ccac198bc06adda4a76ff67eb8d18a25">
                <ds:Transforms>
                    <ds:Transform Algorithm="http://www.w3.org/2000/09/xmldsig
```

(continues on next page)

(continued from previous page)

```
↪#enveloped-signature"/>
                    <ds:Transform Algorithm="http://www.w3.org/2001/10/xml-exc-c14n
↪#"/>
                </ds:Transforms>
                <ds:DigestMethod Algorithm="http://www.w3.org/2001/04/xmlenc#sha256
↪"/>
                <ds:DigestValue>1dFmdN5RyX9MpzoWNdn7H0UvYHdACURaVVeIHVQ/4SQ=</
↪ds:DigestValue>
            </ds:Reference>
        </ds:SignedInfo>
        <ds:SignatureValue>...</ds:SignatureValue>
        <ds:KeyInfo>
            <ds:X509Data>
                <ds:X509Certificate>...</ds:X509Certificate>
            </ds:X509Data>
        </ds:KeyInfo>
    </ds:Signature>
    <samlp:Status>
        <samlp:StatusCode Value="urn:oasis:names:tc:SAML:2.0:status:Success"/>
    </samlp:Status>
    <saml:Assertion xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                    xmlns:xs="http://www.w3.org/2001/XMLSchema"
                    ID="_f0579590dbcee44e77ca6f5b15924d446de19e5e41"
                    Version="2.0"
                    IssueInstant="2022-02-01T12:30:02Z">
        <saml:Issuer>https://ucs-sso.school2.intranet/simplesamlphp/saml2/idp/
↪metadata.php</saml:Issuer>
        <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
            <ds:SignedInfo>
                <ds:CanonicalizationMethod Algorithm="http://www.w3.org/2001/10/
↪xml-exc-c14n#"/>
                <ds:SignatureMethod Algorithm="http://www.w3.org/2001/04/xmldsig-
↪more#rsa-sha256"/>
                <ds:Reference URI="#_f0579590dbcee44e77ca6f5b15924d446de19e5e41">
                    <ds:Transforms>
                        <ds:Transform Algorithm="http://www.w3.org/2000/09/xmldsig
↪#enveloped-signature"/>
                        <ds:Transform Algorithm="http://www.w3.org/2001/10/xml-exc-
↪c14n#"/>
                    </ds:Transforms>
                    <ds:DigestMethod Algorithm="http://www.w3.org/2001/04/xmlenc
↪#sha256"/>
                    <ds:DigestValue>wEtCbO7niag4S3jSyksx2v1oMN/ZXsemkngzug8aSrc=</
↪ds:DigestValue>
                </ds:Reference>
            </ds:SignedInfo>
            <ds:SignatureValue>...</ds:SignatureValue>
            <ds:KeyInfo>
                <ds:X509Data>
                    <ds:X509Certificate>...</ds:X509Certificate>
                </ds:X509Data>
            </ds:KeyInfo>
        </ds:Signature>
        <saml:Subject>
            <saml:NameID SPNameQualifier="https://login.keycloak.idbroker.intranet/
↪auth/realms/ID-Broker/broker/ExampleSA2/endpoint/descriptor"
                         Format="urn:oasis:names:tc:SAML:2.0:nameid-format:transient
↪">_b5c01dce5bcb1c6df188f00994452edb2ac344ff8c</saml:NameID>
            <saml:SubjectConfirmation Method="urn:oasis:names:tc:SAML:2.0:cm:bearer
↪">
                <saml:SubjectConfirmationData NotOnOrAfter="2022-02-01T12:35:02Z"
```

(continues on next page)

(continued from previous page)

```
                                                Recipient="https://login.keycloak.
↪idbroker.intranet/auth/realms/ID-Broker/broker/ExampleSA2/endpoint"
                                                InResponseTo="ID_cc9da054-144b-4c42-
↪8dbc-2008860f2f01"/>
            </saml:SubjectConfirmation>
        </saml:Subject>
        <saml:Conditions NotBefore="2022-02-01T12:29:32Z" NotOnOrAfter="2022-02-
↪01T12:35:02Z">
            <saml:AudienceRestriction>
                <saml:Audience>https://login.keycloak.idbroker.intranet/auth/
↪realms/ID-Broker/broker/ExampleSA2/endpoint/descriptor</saml:Audience>
            </saml:AudienceRestriction>
        </saml:Conditions>
        <saml:AuthnStatement AuthnInstant="2022-02-01T12:30:02Z"
                             SessionNotOnOrAfter="2022-02-02T00:30:02Z"
                             SessionIndex="_
↪76728957c11b9423982865245e078b2555e7478eeb">
            <saml:AuthnContext>
                <saml:AuthnContextClassRef>urn:oasis:names:tc:SAML:2.
↪0:ac:classes:PasswordProtectedTransport</saml:AuthnContextClassRef>
            </saml:AuthnContext>
        </saml:AuthnStatement>
        <saml:AttributeStatement>
            <saml:Attribute Name="entryUUID"
                            NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-
↪format:uri">
                <saml:AttributeValue xsi:type="xs:string">602ac394-17a6-103c-89a6-
↪49b4f56b1bc0</saml:AttributeValue>
            </saml:Attribute>
        </saml:AttributeStatement>
    </saml:Assertion>
</samlp:Response>
```

16. redirect:

TODO description

```
https://login.keycloak.idbroker.intranet/auth/realms/ID-Broker/login-actions/first-
↪broker-login?
 client_id=python-client&
 tab_id=qt_dqhNpMEY
```

17. redirect:

TODO description

```
https://login.keycloak.idbroker.intranet/auth/realms/ID-Broker/broker/after-first-
↪broker-login?
 session_code=KHfAMlxOVSIyISKwo4EuK4l--gTxk6QkCe26dI-UCS8&
 client_id=python-client&
 tab_id=qt_dqhNpMEY
```

18. redirect: service provider

Now, with the correct SAML ticket being sent to Keycloak the browser is authenticated. From the OAuth2 point of view this has happened "behind the scenes". The OAuth2 flow continues with a session, and an auth code, and a redirect to the *testapp*.

```
Location: http://10.205.2.110:5000/oidc_callback?
␣
↪state=eyJjc3JmX3Rva2VuIjogImliT19ncWVXRFdNYnFtTUxnNUczZmVjM3BjLXREYnRXIiwgImRlc3RpbmF0aW9uIjogIn
↪%3D%3D&
 session_state=e31ac91a-a628-425f-9b36-aece4a1ce2fd&
 code=bbd3bdc4-f5dd-486b-bcd7-e2b70823cadb.e31ac91a-a628-425f-9b36-aece4a1ce2fd.
↪343f8446-c6e1-4911-b890-c37a52fd47e9
```

19. request

The browser follows the redirect to the *testapp*, delivering the auth code.

```
http://10.205.2.110:5000/oidc_callback?
␣
↪state=eyJjc3JmX3Rva2VuIjogImliT19ncWVXRFdNYnFtTUxnNUczZmVjM3BjLXREYnRXIiwgImRlc3RpbmF0aW9uIjogIn
↪%3D%3D&
 session_state=e31ac91a-a628-425f-9b36-aece4a1ce2fd&
 code=bbd3bdc4-f5dd-486b-bcd7-e2b70823cadb.e31ac91a-a628-425f-9b36-aece4a1ce2fd.
↪343f8446-c6e1-4911-b890-c37a52fd47e9
```

20. request

The *testapp* now wants to exchange the auth code for an access token and an id token. So the *testapp* queries Keycloak directly to do just that.

```
POST https://login.keycloak.idbroker.intranet/auth/realms/ID-Broker/protocol/
↪openid-connect/token
```

21. response

The *testapp* actually gets three tokens - the access token, the id token and also a refresh token, which allows it to refresh the access once the access token has expired.

```
{
    "access_token":
↪"eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSldUIiwia2lkIiA6ICJuRHBjM2szTkFzZGc4YndJbGRZRnk5MjVQYWxBR2g5bDI
↪eyJleHAiOjE2NDM4MDE1ODQsImlhdCI6MTY0MzgwMTI4NCwiYXV0aF90aW1lIjoxNjQzNzk3MjkyLCJqdGkiOiIzNjk2OTJm
↪tip2fcB6it5EcMfUXRbEoZfvc9ofpBjbrGXQJHD6yXYyWnImGJsWBd8-
↪6wfukxwIu433sVCIFy5k0bYkyqO9OHDs5RZJ4o6FTN8EEy4rN1Ne0r4WR_3kAoJPxiQMrC9K-6_
↪1GgtsOQyAbQns74zVeBj2zis1JlcusSNyIDidQ6beqsVBQalfdgnzrhejvh3LnVjPOedNtrFurUdbZ-
↪JIsWsv3uwo_OX6ZrtvUCvBzNEFT85gmq1uPqFTk4stW7jpVZSc5-
↪UKMcFNjVsMCHTDRaTffIL2WiUVPWQh239M7Q4NSPN7JBDm_
↪pMBtowOBgVEAda9YeXDgJUlrqc2cYjdZGX1GQ6dGtApeVCiCOv6OpmhFQmT4-fKKmpqatc_
↪Aia5NinJ21PstJDRIXe4F8T43KYtfAwxjwAcMxGzyiuvdDphrbDDTNxBsUH7IFDUw1QCyuzUavus3r8vw0z9XOiNINq_
↪-BuXnQrC0fkPLhqzshFnZGFJ8vJO24Fn2vhkaSh3CKctfusSEB1SRA9Kl2icT5BRZ_bNeEvyXk-hzmZ_
↪NhCiVdOPfIw4i_Cfb8KRd6BW0Nk68BSmWdTV1AAFRF3x9UB5sEOZooLK4gYwkX0kEduZ-
↪aGxK6t2Tz45XLCR_vsTonKe7o6F5D47b-VVhm4N8PQHQgq28Bd8VCZlyTqipUU",
    "expires_in": 300,
    "refresh_expires_in": 1800,
    "refresh_token":
```

(continues on next page)

(continued from previous page)


```
↪"eyJhbGciOiJIUzI1NiIsInR5cCIgOiAiSldUIiwia2lkIiA6ICI3ZWEwMDYzNi0yZTBlLTQ0YzMtOGRhM$1kMGM1M2U5ZTc
↪eyJleHAiOjE2NDM4MDMwODQsImlhdCI6MTY0MzgwMTI4NCwianRpIjoiMmZhNjdmZGUtNzUwNy00NDYzLTlhYzAtZGJmMDI
↪DkTZc2H_yqN8QJPV4YuUk1tqdbdpzCd56sCKZ7pDDQQ",
    "token_type": "Bearer",
    "id_token":
↪"eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSldUIiwia2lkIiA6ICJuRHBjM2szTkFzZGc4YndJbGRZRnk5MjVQYWxBR2g5bDI
↪eyJleHAiOjE2NDM4MDE1ODQsImlhdCI6MTY0MzgwMTI4NCwiYXV0aF90aW1lIjoxNjQzNzk3MjkyLCJqdGkiOiJhODBiZjh
↪gVaJTVNwWwNRh04aat9BV8ObWaVuGZtSlOMybLktzwERjrG8TBJ8wpk9rWzaym7vl34QJ30cLmCLdLgHHXN_
↪rYpLiAkE6_L6sjfIwPqPHSCzMFCtohpnwdwHI-
↪tA0ii3eR6ms4EqutXNVrZNluR3ApI9GW8owGEit4SQQstBm_
↪qwYYMR5dfkAuIxh24g1U2NvGO5ZrtBU2o7zc0M26hgWQ6y9MAaMa-PYaVecK-u21d_MkWbccm2Kg-
↪j2ErNDX4N_1l6WELdZjppzSnH-zYY-TV7FbhV_jgS2FPoVt3SDOhsH6CBjxm4J5t48Vowjlf0vk0V6lm_
↪ys5batZ5Z1fvcd9-DcY8RjmHuvU_T5ocKrrK2dKcKNnjrnBZUiwQzoszjNY9t02HLmAI8mfL5nzyO-
↪OM6jX9YDa0jy_6QkeInQXN4Dgh2YNd333F9lPrnJph5oul2V9-
↪5lV6scD2hPRSwrnPVBAsVcW14WA2nttvYEQoJ-J637oXokP2XCqftcd5zYFV3mjzDEyuH0GMvkO8kpK6_
↪4s2hh9KFDzeIWTRoFfSbLhq5b_rpSasGzEb3tfR20YlCTAQo1j06iBsAkBdebopn8TKU_
↪eLFpExxc90pXOooNmznWGmZT36buIUQJIfiJh0C2WIWvU7au_gVKzU9cFuu54F2xUmQA97JEKsDCU",
    "not-before-policy": 0,
    "session_state": "134eae83-970a-4d3a-84a5-a8e331be7d22",
    "scope": "openid email profile"
}
```


Overview over some 3letter abbreviations

Before we look at the details of the tokens, here are some of the abbreviations used in the tokens.

Table 6.1: Abbreviations used in tokens

Field	example	name	details
acr	`0`	Authentication Context Class Reference	0: identified by session cookie, 1: fresh login with username & password, 2: fresh login with username & password & second factor
alg	`RS256`	algorithm used to sign the JWT token	
aud	`account`	Audience	Audience(s) that this ID Token is intended for, e.g. Bettermarks. Can be more then one!
azp	`python-client`	Authorized party	The party to which the ID Token was issued. This Claim is only needed when the ID Token has a single audience value and that audience is different than the authorized party
exp	`1643801311`	Expiration time	secs since epoch
iat	`1643801011`	Issued At	secs since epoch
iss	`https://login.keycloak.idbroker.intranet/auth/realms/ID-Broker`	Issuer	
jti	`975616e5-f847-43c8-a6a1-`	JWT ID	can be used to prevent reuse of the token
kid	`nDpc3k3NAsdg8bwIldYFy925Havv0`	Key Identifier	
scope	`openid email profile`	Scope Values	
sub	`f:d7e4ce14-cf07-4e1f-803`	Subject	A locally unique and never reassigned identifier within the Issuer for the End-User, which is intended to be consumed by the Client `b1bc0`
typ	`Bearer`	Type	media type of this complete JWT

More details:

- https://darutk.medium.com/understanding-id-token-5f83f50fa02e
- https://datatracker.ietf.org/doc/html/rfc7519#section-4.1
- https://www.iana.org/assignments/jwt/jwt.xhtml
- https://openid.net/specs/openid-connect-core-1_0.html#IDToken

access token

Details of the access token

```
{
    "header": {
        "alg": "RS256",
        "typ": "JWT",
        "kid": "nDpc3k3NAsdg8bwIldYFy925PalAGh9l3gb2qeHavv0"
    },
    "payload": {
        "exp": 1643801311,
        "iat": 1643801011,
```

(continues on next page)

(continued from previous page)


```
        "auth_time": 1643797292,
        "jti": "975616e5-f847-43c8-a6a1-9e77d97a401c",
        "iss": "https://login.keycloak.idbroker.intranet/auth/realms/ID-Broker",
        "aud": "account",
        "sub": "f:d7e4ce14-cf07-4e1f-803c-de996dd24da7:5bf644e6-178e-103c-8451-
↪49b4f56b1bc0",
        "typ": "Bearer",
        "azp": "python-client",
        "session_state": "134eae83-970a-4d3a-84a5-a8e331be7d22",
        "acr": "0",
        "allowed-origins": [
            ""
        ],
        "realm_access": {
            "roles": [
                "offline_access",
                "default-roles-id-broker",
                "uma_authorization"
            ]
        },
        "resource_access": {
            "account": {
                "roles": [
                    "manage-account",
                    "manage-account-links",
                    "view-profile"
                ]
            }
        },
        "scope": "openid email profile",
        "sid": "134eae83-970a-4d3a-84a5-a8e331be7d22",
        "email_verified": false,
        "name": "student one one",
        "preferred_username": "5bf644e6-178e-103c-8451-49b4f56b1bc0",
        "given_name": "student one",
        "family_name": "one"
    },
    "signature": "MRg4+jDWPQyK5F0egZfNpD6IU5VJ20IDUTerywCpvJ4Uwr8/
↪DhBZqsmSL8EO7As4ZBgP0gZjHvGkYOrx90MtCxdzF8ygNauIMV6/
↪rhdzfynL3mJv12qza1iT3QcW3wNlxF3t/
↪2oUBEfQ5Jtm2SJkAjjZ3Nm3EmtiqWboiVEykzPPrpdWeixBt9UldYb2nU/
↪ME7XuCcFcQ98O7exB0dom4IMf+74O22ETCeBObIrIFeHfrSU1f2uZOLkwwFSKpa+wFeLln4K2GoO63F4RIdlWsOnSVj+fZt
↪hSsDcQAnS5KxlJRysVcr8JqjliVpZDgTAh5NNNff/CXyZDe6/pXRf3i6K+2nlPMxqpDs1hdKdDti8/
↪Lyf+P9NFz7WlTkddueTA5NFECGzhCPXgZXDo6qn0I1ASs3+bnQNhD5zUEDTUYGq4OclJAQbwENiI3WxrunHAlIFiuqDSZPE
↪BgRq7okF7aBAVGDIHqr09rmaSGgb8KTp4ooKw9SPY1ki4PBW2fQMs9PNTmNcmTknLO5E1h7rBF/ByaE="
}
```


ID Token

Details of the id token.


```
{
    "header": {
        "alg": "RS256",
        "typ": "JWT",
        "kid": "nDpc3k3NAsdg8bwIldYFy925PalAGh9l3gb2qeHavv0"
    },
    "payload": {
        "exp": 1643801311,
```


(continues on next page)

(continued from previous page)


```
        "iat": 1643801011,
        "auth_time": 1643797292,
        "jti": "b2be83a3-5064-4d72-8d34-bc4150976cdc",
        "iss": "https://login.keycloak.idbroker.intranet/auth/realms/ID-Broker",
        "aud": "python-client",
        "sub": "f:d7e4ce14-cf07-4e1f-803c-de996dd24da7:5bf644e6-178e-103c-8451-
↪49b4f56b1bc0",
        "typ": "ID",
        "azp": "python-client",
        "session_state": "134eae83-970a-4d3a-84a5-a8e331be7d22",
        "at_hash": "lj9X-mi3DZFlQsxzwdFecw",
        "acr": "0",
        "sid": "134eae83-970a-4d3a-84a5-a8e331be7d22",
        "email_verified": false,
        "name": "student one one",
        "preferred_username": "5bf644e6-178e-103c-8451-49b4f56b1bc0",
        "given_name": "student one",
        "family_name": "one"
    },
    "signature": "eHEx/
↪LtqtraYg3mn1UYSWzaBIpIoz7g05svjqzvsxtvEsL+5PhSUPfNlmYreL5CfALU5ev9A0PMrMvLiapTMtwCVu2KxddLzBh6H
↪A6UDeknfkcfXsPO/
↪Smd6lDmBqoTCjJsv4FrPCquWlBsfTNWy9ZINqo0i0l9k6AOFzEObDDNkvJcJevYxPtuTDxa9RBQ49cRRoF7aAZuua7Gd5Q4
↪VY7rpeDn2hnkCiklqPPWgKlQzPNpo9z3D8jUJ3hGKakP7dqbPdeJy/
↪Kqw0Zwh0IKhdFUnaoUmg3VmdqUPNDTsZfPfT8NMATL7HeDzuqrFH142Bw15WEuQWsZGguLSJcC4HmGvJtvmKv0WT6Un76M+
↪"
}
```


Refresh Token

And the details of the refresh token.


```
{
    "header": {
        "alg": "RS256",
        "typ": "JWT",
        "kid": "nDpc3k3NAsdg8bwIldYFy925PalAGh9l3gb2qeHavv0"
    },
    "payload": {
        "exp": 1643801311,
        "iat": 1643801011,
        "auth_time": 1643797292,
        "jti": "b2be83a3-5064-4d72-8d34-bc4150976cdc",
        "iss": "https://login.keycloak.idbroker.intranet/auth/realms/ID-Broker",
        "aud": "python-client",
        "sub": "f:d7e4ce14-cf07-4e1f-803c-de996dd24da7:5bf644e6-178e-103c-8451-
↪49b4f56b1bc0",
        "typ": "ID",
        "azp": "python-client",
        "session_state": "134eae83-970a-4d3a-84a5-a8e331be7d22",
        "at_hash": "lj9X-mi3DZFlQsxzwdFecw",
        "acr": "0",
        "sid": "134eae83-970a-4d3a-84a5-a8e331be7d22",
        "email_verified": false,
        "name": "student one one",
        "preferred_username": "5bf644e6-178e-103c-8451-49b4f56b1bc0",
        "given_name": "student one",
        "family_name": "one"
    },
```


(continues on next page)

(continued from previous page)

```
    "signature": "eHEx/
↪LtqtraYg3mn1UYSWzaBIpIoz7g05svjqzvsxtvEsL+5PhSUPfNlmYreL5CfALU5ev9A0PMrMvLiapTMtwCVu2KxddLzBh6He
↪A6UDeknfkcfXsPO/
↪Smd6lDmBqoTCjJsv4FrPCquWlBsfTNWy9ZINqo0i0l9k6AOFzEObDDNkvJcJevYxPtuTDxa9RBQ49cRRoF7aAZuua7Gd5Q4
↪VY7rpeDn2hnkCiklqPPWgKlQzPNpo9z3D8jUJ3hGKakP7dqbPdeJy/
↪Kqw0Zwh0IKhdFUnaoUmg3VmdqUPNDTsZfPfT8NMATL7HeDzuqrFH142Bw15WEuQWsZGguLSJcC4HmGvJtvnKv0WT6Un76M+
↪"
}
```

22. request student_details

Now the *testapp* (a.k.a as client in OAuth2 terms) could do something magical with the access token, e.g. ask the resource server for student details. We haven't documented this step, but one thing is required: the access token needs to be passed along with the request.

23. response

Before fetching data the resource server (e.g. the Self-disclosure API) would need to validate the signature of the access token.

```
import jwt

# see step 19
access_token =
↪"eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSldUIiwia2lkIiA6ICJuRHBjM2szTkFzZGc4YndJbGRZRnk5MjVQYWxBR2g5bDI
↪eyJleHAiOjE2NDM4MDEzMTEsImlhdCI6MTY0MzgwMTAxMSwiYXV0aF90aW1lIjoxNjQzNzk3MjkyLCJqdGkiOiI5NzU2MTZi
↪MRg4-jDWPQyK5F0egZfNpD6IU5VJ20IDUTerywCpvJ4Uwr8_
↪DhBZqsmSL8EO7As4ZBgP0gZjHvGkYOrx90MtCxdzF8ygNauIMV6_
↪rhdzfynL3mJv12qza1iT3QcW3wNlxF3t_
↪2oUBEfQ5Jtm2SJkAjjZ3Nm3EmtiqWboiVEykzPPrpdWeixBt9UldYb2nU_
↪ME7XuCcFcQ98O7exB0dom4IMf-74O22ETCeBObIrIFeHfrSU1f2uZOLkwwFSKpa-
↪wFeLln4K2GoO63F4RIdlWsOnSVj-
↪fZtzRYDHzbXIiVErFj7ig4tAXLDQgfBp8nE1l0K3COOghnpiLhutocAU17yKaZuxiHeP7LOJxlG43SdGpo6S3AqeGIbd0coe
↪hSsDcQAnS5KxlJRysVcr8JqjliVpZDgTAh5NNNff_CXyZDe6_pXRf3i6K-2nlPMxqpDs1hdKdDti8_
↪Lyf-P9NFz7WlTkddueTA5NFECGzhCPXgZXDo6qn0I1ASs3-
↪bnQNhD5zUEDTUYGq4OclJAQbwENiI3WxrunHAlIFiuqDSZPEtse4U8vLtAjV1Gp_
↪BgRq7okF7aBAVGDIHqr09rmaSGgb8KTp4ooKw9SPY1ki4PBW2fQMs9PNTmNcmTknLO5E1h7rBF_ByaE"

# This is the public key from Keycloak, from  https://login.keycloak.idbroker.
↪intranet/auth/realms/ID-Broker
keycloak_public_key =
↪"MIICIjANBgkqhkiG9w0BAQEFAAOCAg8AMIICCgKCAgEA6hjhC3946VIzORQ7SLRVznM7ei0CEDFSRnPy63PwzUeIspBHwU
↪BKmjZo2u+YCEumpDMg+2rarzHFyFLkXAOsR9kQkbvrDGnOJz5cpjn0m2lp4KJ/
↪y4+oKAoer3YQxDWDwplPsCTv3wsPF/u5p8P5/
↪QJElesxwkMQEsGGqii4yduBZ+O4a1NzF7l+WGeYtjtdeHkcgzJfjbrh7orP/
↪7Eoqtb+5LT1alSRbH3ejstNjS+OrnTCS+jZ5+aQdGn3TO+sNDgw2FIIWG4USCteBiLgr+GK6X9YFv3wX+7lDGVjGul+DDjn
↪ln1wdd/
↪OFEetEBTkETGcNyz8hvSBCAQ8BaqWe40gFlMKs9FcjfI4BTFjj92oaoZixlCn+vJ6anbA0vWS952RsQSEvxjK4N77PaoKfL
↪2GDIUeCkYNIjmSOBAnC8SPOA0rsEjT3rE37meWgiKfG57YUFA/
↪8aYipXtiPufKBhvPBL+MAM+ZW8Rf2RioyVE7FoRcCAwEAAQ=="

# The key needs to have a header and a footer...
keycloak_with_headers = "-----BEGIN PUBLIC KEY-----\n" + keycloak_public_key + "\n-
↪----END PUBLIC KEY-----"

# ... and can then be verified directly.
# Once can disable the expiration date verification, when debugging a session␣
↪later on.
```

(continues on next page)

(continued from previous page)

```
# Of course, you wouldn't do this in production.
verified_token = jwt.decode(
    access_token,
    key=keycloak_with_headers,
    audience='account',
    algorithms=['RS256'],
    # options=dict(verify_exp=False) # disable expiration checking
)
```

24. follow redirect URL

The user follows the redirect to the protected resource.

```
http://10.205.2.110:5000/private
```

25. content

Which she can now access because of the id_token in the content of `/private`.

6.1.4 Alternatives

We have thought about alternatives, which we note here.

Auth code flow II - First SAML

First SAML, then OIDC - doesn't work:

1. no login session with Keycloak (bomb in step 5). Could be worked around using a mini login app
2. if the Keycloak session expires and somehow the service provider needs to restart the OIDC session, they don't have the IDP information and hence can't redirect automatically to the right SAML login server

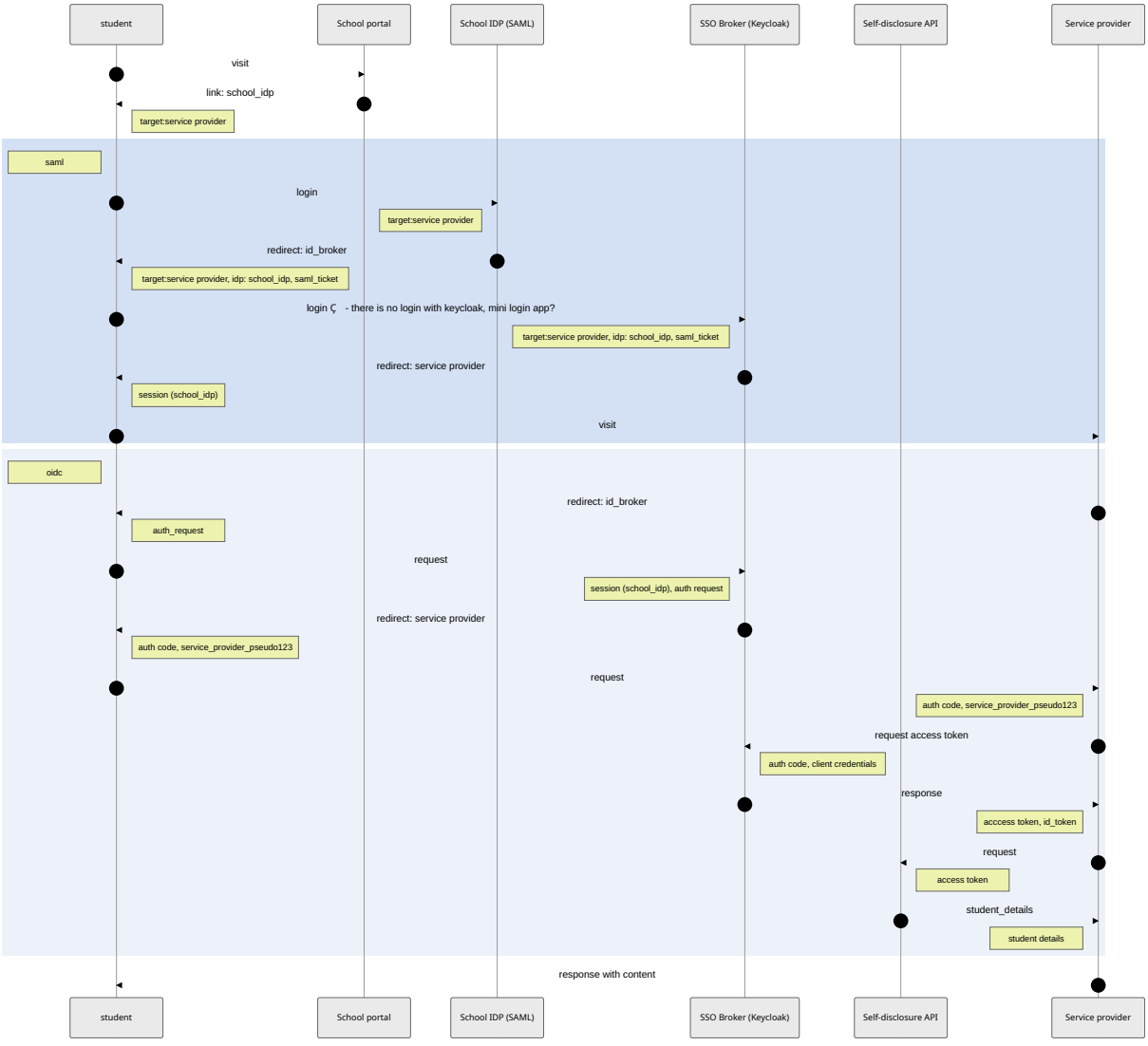
Client credentials flow

This flow would give the client application (service provider) complete access to all student data on the resource server. We hope that we won't need it.

6.2 Data model

Users, school classes and schools each have been extended by the following attributes:

- `ucsschoolRecordUID` saves the `entryUUID` of the object on source system (school authority).
- `ucsschoolSourceUID` saves the name of the school authority.
- `idBrokerPseudonym0001` - `idBrokerPseudonym0030` save service provider specific pseudonyms.


student
School portal
School IDP (SAML)
SSO Broker (Keycloak)
Self-disclosure API
Service provider
visit
link: school_idp
target:service provider
saml
login
target:service provider
redirect: id_broker
target:service provider, idp: school_idp, saml_ticket
login Ç - there is no login with keycloak, mini login app?
target:service provider, idp: school_idp, saml_ticket
redirect: service provider
session (school_idp)
visit
oidc
redirect: id_broker
auth_request
request
session (school_idp), auth request
redirect: service provider
auth code, service_provider_pseudo123
request
auth code, service_provider_pseudo123
request access token
auth code, client credentials
response
acccess token, id_token
request
access token
student_details
student details
response with content
student
School portal
School IDP (SAML)
SSO Broker (Keycloak)
Self-disclosure API
Service provider

6.2.1 Mapping LDAP / UDM / UCS@school attributes

As written in *ID Broker components* (page 15) the *Self-disclosure API* and the *Provisioning API* use the *Kelvin REST API* to access user / group data. The Kelvin REST API exposes the *UCS@school library* models in its API and uses the *UDM REST API* to access the IDMs LDAP database.

The UCS@school library, the UDM REST API and OpenLDAP target different scenarios / layers and thus have different data models and use slightly different names for the same attributes. Below are tables that should help navigate the different data layers.

Users

Table 6.2: Users mapping of LDAP attribute → UDM property → UCS@school attribute

LDAP attribute	UDM property	UCS@school attribute	Example
uid	username	name	demo_student
entryUUID			4e2d101a-b843-48d0-81d3-68a74940adc7
givenName	firstname	firstname	Alice
mailPrimaryAddress	mailPrimaryAddress	email	first.last\@example.com
sn	lastname	lastname	Bauer
ucsschoolRole	ucsschoolRole	ucsschool_role	student:school:DEMOSCHOOL
ucsschoolRecordUID	ucsschoolRecordUID		4e2d101a-b843-48d0-81d3-68a74940adc7
ucsschoolSourceUID	ucsschoolSourceUID		school authority name
idBrokerPseudonym00XX	idBrokerPseudonym00XX		4e2d101a-b843-48d0-81d3-68a74940adc7

Groups

Table 6.3: Groups mapping of LDAP attribute → UDM property → UCS@school attribute

LDAP attribute	UDM property	UCS@school attribute	Example
cn	name	name	DEMOSCHOOL-Democlass
description	description		Math work group
ucsschoolRole	ucsschoolRole	ucsschool_role	school_class:school:DEMOSCHOOL
uniqueMember	users	users	['uid=demo_student,cn=schueler,cn=...', 'uid=demo_teacher,...']
ucsschoolRecordUID	ucsschoolRecordUID		4e2d101a-b843-48d0-81d3-68a74940adc7
ucsschoolSourceUID	ucsschoolSourceUID		school authority name
idBrokerPseudonymC	idBrokerPseudonymC		4e2d101a-b843-48d0-81d3-68a74940adc7

Schools

Table 6.4: Schools mapping of LDAP attribute → UDM property → UCS@school attribute

LDAP attribute	UDM property	UCS@school attribute	Example
ou	name	name	DEMOSCHOOL
displayName	displayName	display_name	Demo School
ucsschoolRole	ucsschoolRole	ucsschool_role	school:school:DEMOSCHOOL
ucsschoolRecordUID	ucsschoolRecordUID		4e2d101a-b843-48d0-81d3-68a74940adc7
ucsschoolSourceUID	ucsschoolSourceUID		school authority name
idBrokerPseudonym00XX	idBrokerPseudonym00XX		4e2d101a-b843-48d0-81d3-68a74940adc7

6.3 manage-service-providers

The tool `manage-service-providers` is used to add service providers and generate pseudonyms for existing users, school classes and schools. This document describes how the script works internally. Visit Backup - Provisioning API[15] for information about how to use it.

The tool adds a mapping of the service provider name to one of the UDM properties in the set of `idBrokerPseudonym0001` to `idBrokerPseudonym0030`, in which the corresponding pseudonym is saved. The first property, which hasn't been added to the mapping, is chosen. The tool also generates a salt and saves it as another mapping (service provider to salt). Both are saved in a `settings/data` object. The values are protected by ACLs and can be read/ written by the groups `id-broker-settings-secrets-read` and `id-broker-settings`, which are created during the installation process.

After that, the script iterates over all existing users, groups and school and generates a pseudonym using the salt of the service provider, the name of the school authority as well as the entry uuid of the object on school authority side:

```
hash(service_provider_salt, entry_uuid , school_authority)
```

We save the `entry_uuid` inside `ucsschoolRecordUID` and the `school_authority` inside `ucsschoolSourceUID` for each user, school class and school.

[15] https://univention.gitpages.knut.univention.de/id-broker/operations-manual/installation.html#install-id-broker-backup-provisioning

CHAPTER

SEVEN

GLOSSARY

Identity Provider (IDP)
: Instance that provides information to authenticate and authorize identities. In case of ID Broker scenarios typically an SAML or OpenID Connect IDP hosted by a *school authority*.

Provisioning API
: REST API of the ID Broker which is used by *school authorities* to send pseudonyms and a limited set of meta information on users and groups to the ID Broker.

School Authority
: In context of this document *school authority* subsumes the various institutions which serve one or several schools with IT infrastructure. That includes that the school authority holds the identity store for all learners and teachers of an environment. This can be a single School, a school authority with several schools, or an environment hosting services for a federal state. Typically these are environments hosting a UCS@school domain.

Self-disclosure API
: REST API of the ID Broker which allows retrieval of meta information of an authorized user (focus is role of the user and the assigned learning groups). The API is derived from an API introduced by *Bettermarks* and sometimes referred to as *Bettermarks API*.

Service Provider (SP)
: Instance that provides a service that is configured for a single sign-on with the ID Broker, typically content providers or applications for pupils and teachers.

SSO Broker
: The main job of the SSO Broker component is to handle multiple-tenant authentication, using pseudonyms. This involves the student (or her browser) doing the login and passing authentication tokens/tickets back and forth.

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- search

INDEX

I

P

S