univention
be open.

UCS@school ID Connector

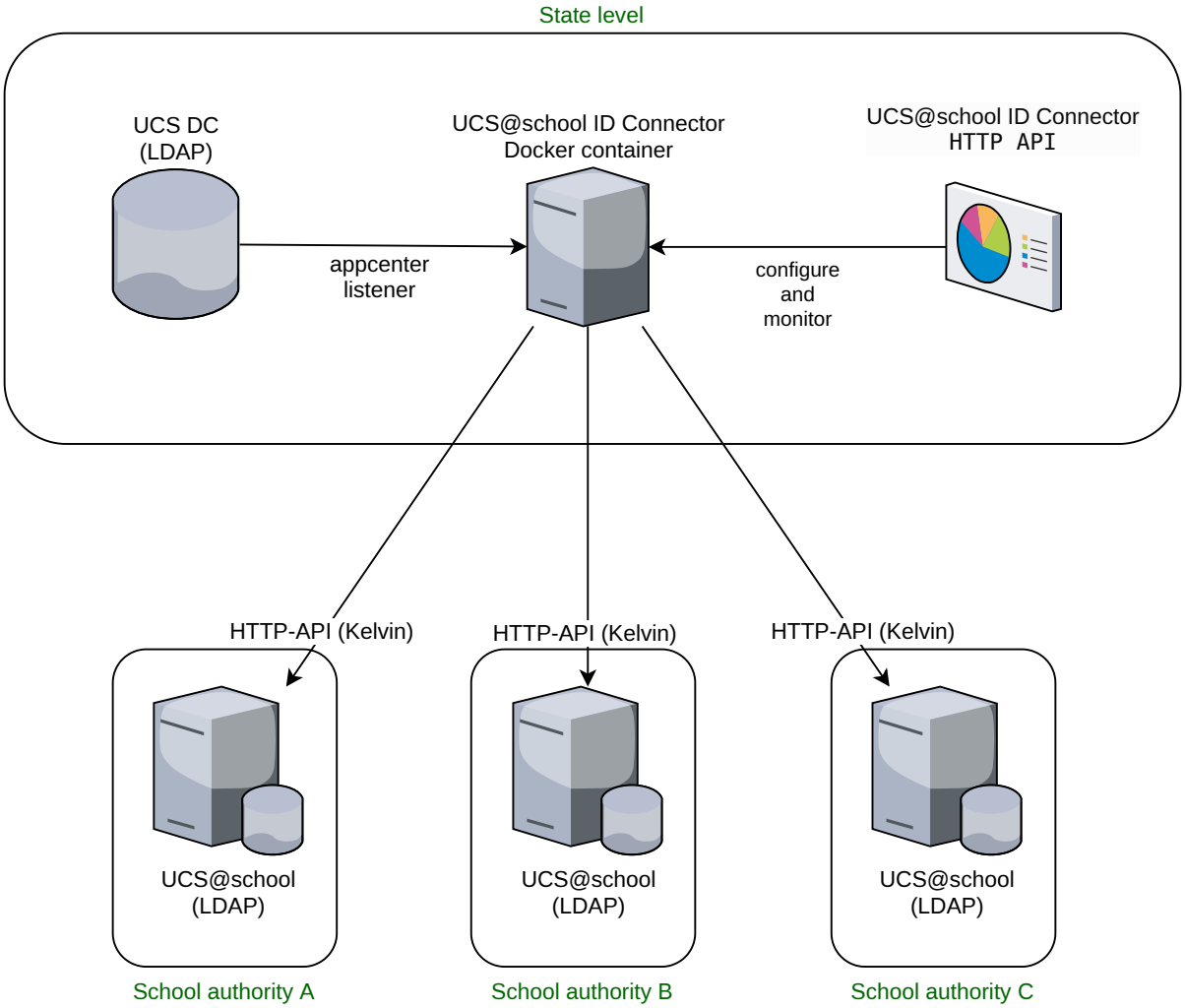
Release 2.2.4

Univention GmbH

Dec 08, 2022

CONTENTS

License AGPL v3 [1] python 3.8 [2] The ID Connector connects an UCS@school directory to any number of other UCS@school directories (1:n). It is designed to connect state directories with school districts, but can also be used in other contexts. The connection takes place unidirectional: user data (user, school affiliation, class affiliations) is transferred from a central directory (e.g. country directory) to district or school directories. Prerequisite is the use of the *UCS@school Kelvin* API on the school authorities. For this a configuration is necessary in advance to create an assignment "Which school users should be transferred to which remote instance?" Then these users are created, updated and deleted.



In this documentation, you will learn how to administer an ID Connector setup, and we hope to teach you how to develop plugins for ID Connector as well.

Note: At the moment, the ID Connector setup is only used in German-speaking countries. Hence, you will encounter a few German terms in this documentation.

Sender An easy one to guess - it actually refers to the sending side of the sync process, which in Germany most likely is a state department.

Traeger This is the organization managing schools. In the ID Connector context it can be thought of as the *recipient* of sync data.

Note: You can use the clipboard icon on the top right of code examples to easily copy the code without python and bash prompts:

[1] https://www.gnu.org/licenses/agpl-3.0
[2] https://www.python.org/downloads/release/python-382/

```
$ echo "hello world"
```

(Hover with your mouse over the code to see the icon)

CHAPTER

ONE

CONTENTS

1.1 Administration

1.1.1 Admin overview

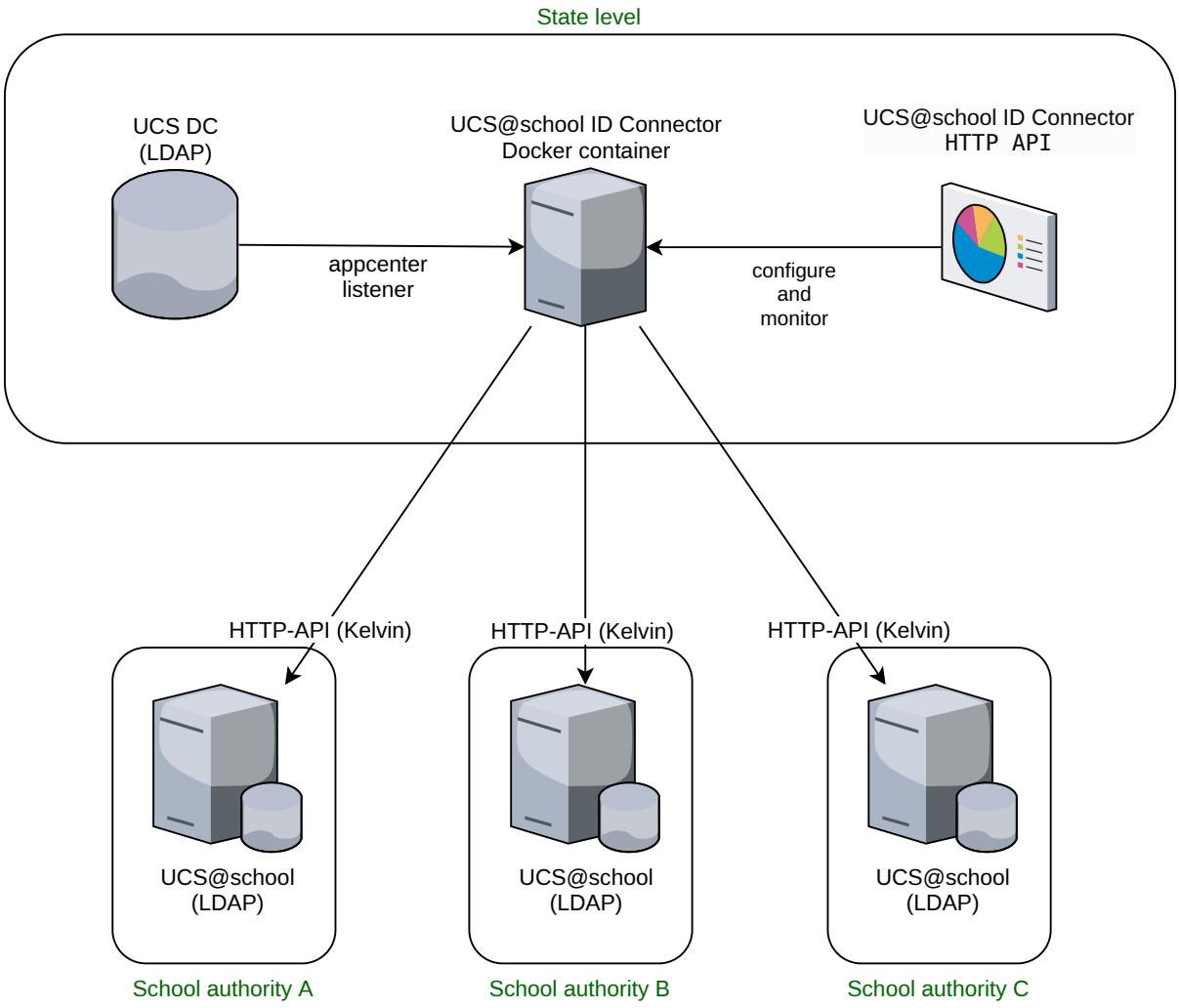


Fig. 1: Simplified overview of the ID Connector

The ID Connector replication system is composed of four components:

1. An *LDAP server* containing user data.

2. A process on the data source UCS server, receiving user creation/modification/deletion events from the LDAP server and relaying them to multiple recipients via HTTP, called the *ID Connector Service*.
3. A process on the data source UCS server to monitor and configure the UCS@school ID Connector service, called the ID Connector *HTTP API*.
4. Multiple recipients of the directory data relayed by the *ID Connector Service*. They run an HTTP-API service, that the *ID Connector Service* pushes updates to.

1.1.2 Admin prerequisites

This administration chapter is useful when you need to administer an ID Connector setup, or you need to integrate ID Connector. To follow this text, you should be familiar with the following aspects of the UCS environment:

LDAP and LDAP listener LDAP is used because it is optimized for reading in a hierarchical structure. It shouldn't be accessed directly, instead *UDM* should be used. OpenLDAP can have plugins, notifier being one of them that is heavily used in UCS. Upon changes in the LDAP directory, the notifier triggers listeners locally and on remote systems.

The listener service connects to all local or remote notifiers in the domain. The listener, when notified, calls listener modules, which are scripts (in shell and python)

You need to be able to:

- understand the basic concepts of LDAP

See also:

See LDAP directory service[3] in *UCS Manual* [1].

Univention Directory Manager Univention Directory Manager (**UDM**) is used for handling user data (and other data) that is stored in the LDAP server, one of two core storage places (the other one is *UCR*). Examples for data are users, roles or machine info. UDM adds a layer of functionality and logic on top of LDAP, hence LDAP shouldn't be used directly, but only through UDM.

You need to be able to:

- understand the concept of UDM
- know the basic structure of UDM objects and their attributes
- add and manage extended attributes

See also:

See Univention Directory Manager (UDM)[4] in *Univention Developer Reference* [2].

Univention Configuration Registry The Univention Configuration Registry (**UCR**) stores configuration variables and settings to run the system, and creates and changes actual Linux configuration files as configured by these variables upon setting said variables.

You need to be able to:

- understand basic UCR concepts
- set and read UCR variables.

See also:

See Administration of local system configuration with Univention Configuration Registry[5] in *UCS Manual* [1].

Univention App Center settings The Univention App Center is an ecosystem similar to the app stores known from mobile platforms like Apple or Google. It provides an infrastructure to build, deploy and run enterprise applications on Univention Corporate Server (UCS). The Univention App Center uses well-known technologies like Docker.

[3] https://docs.software-univention.de/manual/5.0/en/index.html#introduction-ldap-directory-service
[4] https://docs.software-univention.de/developer-reference/5.0/en/udm/index.html#chap-udm
[5] https://docs.software-univention.de/manual/5.0/en/computers/ucr.html#computers-administration-of-local-system-configuration-with-univention-configuration-re

Within the app center, you can configure settings for the individual apps.

See also:

- App settings[6] in *Univention App Center for App Providers* [3]
- Setting of an application in the App Center[7] in *UCS Manual* [1]

UCS@school basics Schools have special requirements for managing what is going on inside them (teachers, students, staff, computer rooms, exams, etc.), and for managing the relation between multiple schools, their operator organizations ("Schulbetreiber"), and possibly ministerial departments above them.

There are several components used within UCS@school, Kelvin (see below) being one of them.

You need to be able to:

- know about UCS@school objects
- know the difference between UCS@school-objects and UDM objects

See also:

- KB 15630 - How a UCS@school user should look like[8]
- KB 16925 - UCS@school work groups and school classes[9]
- UCS@school - Handbuch für Administratoren[10]

UCS@school Kelvin REST API The UCS@school Kelvin REST API (Kelvin) provides HTTP endpoints to create and manage individual UCS@school domain objects like school users, school classes and schools (OUs). It is written in FastAPI, hence in Python 3.

You need to be able to install and configure Kelvin.

See also:

- Overview[11] in *UCS@school Kelvin REST API documentation* [4]
- UCS@school-Objekte im LDAP-Verzeichnisdienst[12] in *UCS@school - Handbuch für Administratoren* [5]

If you want to also develop for the ID Connector, please also see the next chapter *Development* (page 15).

1.1.3 Installation

1.1.3.1 Sending system

The app is available in the Univention App Center. You can install it with:

```
$ univention-app install ucsschool-id-connector
```

This runs the join script `50ucsschool-id-connector.inst`, which creates:

- the file `/var/lib/univention-appcenter/apps/ucsschool-id-connector/conf/tokens.secret` containing the key with which JWT tokens are signed.
- the group `ucsschool-id-connector-admins` (with DN `cn=ucsschool-id-connector-admins,cn=groups,$ldap_base`) who's members are allowed to access the ID Connector HTTP-API.

[6] https://docs.software-univention.de/app-center/5.0/en/configurations.html#app-settings
[7] https://docs.software-univention.de/manual/5.0/en/software/app-center.html#appcenter-configure
[8] https://help.univention.com/t/15630
[9] https://help.univention.com/t/16925
[10] https://docs.software-univention.de/ucsschool-manual/5.0/de/index.html
[11] https://docs.software-univention.de/ucsschool-kelvin-rest-api/overview.html
[12] https://docs.software-univention.de/ucsschool-manual/5.0/de/structure.html#structure-ldap

Use of both is explained later on in *Authentication* (page 8)

Note: You can check the existence of the group with:

```
$ udm groups/group list --filter cn=ucsschool-id-connector-admins
```

Note: Join scripts are registered in LDAP and then executed on any UCS system either before/during/after the join process.

→ KB 13034 - A script shall be executed on each or a certain UCS systems before/during/after the join process[13]

If the above didn't get created, run:

```
$ univention-run-join-scripts --run-scripts --force 50ucsschool-id-connector.inst
```

This forces the (re-)running of the join script.

1.1.3.2 Receiving system

In order for the *ID Connector* app to be able to create/modify/delete users on the target systems an HTTP-API is required on the target system. Currently only the Kelvin API is supported.

Note: This only makes sense if the sender and target systems are in different domains, because in the same domain, users and groups already get synced using other UCS mechanisms.

Install the Kelvin API on each target system:

```
$ univention-app install ucsschool-kelvin-rest-api
```

To allow the *ID Connector* app on the sender system to access the Kelvin-API on the receiving system, it needs an authorized user account. By default, the Administrator account on the receiving system is the only authorized user. To add a dedicated Kelvin API user for the UCS@school ID Connector consult *UCS@school Kelvin REST API documentation* [4] on how to do that.

1.1.4 Configuration

Now that everything is installed, let's configure the setups. We configure the receiving system first, because we need auth credentials used on the receiving system later on the sending system.

1.1.4.1 Configure receiving system - HTTP-API (Kelvin)

You need to install and configure the Kelvin API. This is documented in *UCS@school Kelvin REST API documentation* [4].

We assume that you have a current version of Kelvin installed after reading the documentation.

Note: For the authorization of the *UCS@school ID Connector* at the target system it needs credentials with special privileges. Create a user with the name and password of your choice and add him to the group `ucsschool-kelvin-rest-api-admins`.

[13] https://help.univention.com/t/13034

```
$ udm users/user create --position "cn=users,$(ucr get ldap/base)" \
  --set username=USERNAME-OF-YOUR-CHOICE --set lastname=Kelvin --set firstname=UCS␣
↪\
  --set password="PASSWORD-OF-YOUR-CHOICE"

$ udm groups/group modify --dn "cn=ucsschool-kelvin-rest-api-admins,cn=groups,
↪$(ucr get ldap/base)" \
  --append users="uid=USERNAME-OF-YOUR-CHOICE,cn=users,$(ucr get ldap/base)"
```

Note down the credentials, they are needed for the *school authority configuration on the sending system* (page 10) further down.

WARNING: the password is now in the command history. You might want to delete this using e.g:

```
$ history -d -2
```

After installation and basic configuration you might want to configure mapped UDM properties.

Beyond the standard object properties in UCS@school[14] you can define additional UDM properties that should be available in the Kelvin API on the target system.

For this you would define a configuration in `/etc/ucsschool/kelvin/mapped_udm_properties.json`, e.g.:

```
{
    "user": ["title", "phone", "e-mail"],
    "school": ["description"]
}
```

This would make the listed properties available for the `user` and `school` resources.

Kelvin needs to be restarted after changes to its configuration:

```
$ univention-app restart ucsschool-kelvin-rest-api
```

Note: When configuring Kelvin in detail, remember that the password hashes for LDAP and Kerberos authentication are collectively transmitted in one JSON object to one target attribute. This means it's all or nothing: all hashes are synced, even if empty. You can't select individual hashes.

Note: Please make sure that you configure all the mapped properties that the sending system sends, e.g. `displayName`. If the sender sends more than the receiver is configured to process, you will end up with unexpected errors, e.g. `404` in the log.

1.1.4.2 Configure sending system

The school authorities configuration must be done through the ID Connector *HTTP API*. Do not edit configuration files directly.

[14] https://docs.software-univention.de/ucsschool-kelvin-rest-api/resource-users.html#users-resource-repr

UCS@school ID Connector HTTP API

The HTTP-API of the *ID Connector* app offers three resources:

- *queues*: monitoring of queues
- *school_authorities*: configuration of school authorities
- *school_to_authority_mapping*: configuration of which school we sync to which authority

You can discover the API interactively using one of two web interfaces. They can be visited with a browser at their respective URLs:

- Swagger UI[15]: `https://FQDN/ucsschool-id-connector/api/v1/docs`
- ReDoc[16]: `https://FQDN/ucsschool-id-connector/api/v1/redoc`

The Swagger UI page is especially helpful as it allows sending queries directly from the browser. The equivalent `curl` command lines are then displayed.

An OpenAPI v3 (formerly "Swagger") schema[17] can be downloaded from `https://FQDN/ucsschool-id-connector/api/v1/openapi.json`

Authentication

Only members of the group `ucsschool-id-connector-admins` are allowed to access the HTTP-API.

The user `Administrator` is automatically added to this group for testing purposes. In production, only the regular admin user accounts should be used.

You can authorize yourself in e.g. the Swagger UI using the `Authorize` button.

To use the ID Connector *HTTP API* from a script, a JSON Web Token (JWT)[18] must be retrieved from `https://FQDN/ucsschool-id-connector/api/token`. The token will be valid for a configurable amount of time (default 60 minutes), after which it must be renewed. To change the TTL of the token, open the corresponding *app settings* in the Univention App Center.

Example `curl` command to retrieve a token:

```
$ curl -i -k -X POST --data 'username=Administrator&password=s3cr3t' \
  https://FQDN/ucsschool-id-connector/api/token
```

School authorities mapping

We now need to configure two things:

1. What school authorities do we send data to, and what data can they receive? This is described in this section.
2. What actual schools are handled by which receiving system (school authority)? This is described in the following section: *School to authority mapping* (page 11).

We start with the first mapping, the one for school authorities.

In order to send user data to the target system, it must be decided which properties of which objects to send, and more important, which properties *not* to send.

E.g. there might be telephone numbers for students in the system on the sending side, but those should not be made available to the receiving school system. Instead of forbidding properties, we "map" properties on the sending side to properties on the receiving side.

Here is what the mapping related part of an example configuration looks like:

[15] https://github.com/swagger-api/swagger-ui
[16] https://github.com/Redocly/redoc
[17] https://swagger.io/docs/specification/about/
[18] https://en.wikipedia.org/wiki/JSON_Web_Token

```
...
 {
     "plugin_configs": {
         "kelvin": {
             "mapping": {
                 "users": {
                     "ucsschoolRecordUID": "record_uid",
                     "ucsschoolSourceUID": "source_uid",
                     "roles": "roles"
                 }
             }
         }
     }
 }
...
```

This configures a mapping for the Kelvin plugin that sends the three defined properties to the receiving school:

- The UDM `ucsschoolRecordUID` property should be synced to an UCS@school system as `record_uid`.
- The UDM `ucsschoolSourceUID` property should be synced to an UCS@school system as `source_uid`.
- The *virtual* `roles` property should be synced to an UCS@school system as `roles`

Note: `roles` is *virtual* because there is special handling by the *ID Connector* app mapping `ucsschoolRole` to `roles`.

Warning: When creating users via Kelvin, some attributes are required and therefore have to be present within the mapping:

```
{
  "firstname": "firstname",
  "lastname": "lastname",
  "username": "name",
  "school": "school",
  "schools": "schools",
  "roles": "roles",
  "ucsschoolRecordUID": "record_uid",
  "ucsschoolSourceUID": "source_uid"
}
```

Here is a complete example that you can also find in the section *School authority configuration* (page 33).

```
{
  "name": "Traeger1",
  "url": "https://10.200.3.242/ucsschool/kelvin/v1/",
  "active": true,
  "plugins": [
    "kelvin"
  ],
  "plugin_configs": {
    "kelvin": {
      "username": "Administrator",
      "password": "univention",
      "mapping": {
        "users": {
          "firstname": "firstname",
```

(continues on next page)

(continued from previous page)

```
          "lastname": "lastname",
          "username": "name",
          "disabled": "disabled",
          "mailPrimaryAddress": "email",
          "e-mail": "email",
          "birthday": "birthday",
          "school": "school",
          "schools": "schools",
          "school_classes": "school_classes",
          "title": "title",
          "displayName": "displayName",
          "userexpiry": "expiration_date",
          "phone": "phone",
          "roles": "roles",
          "ucsschoolRecordUID": "record_uid",
          "ucsschoolSourceUID": "source_uid"
        },
        "school_classes": {
          "name": "name",
          "description": "description",
          "school": "school",
          "users": "users"
        }
      },
      "sync_password_hashes": true,
      "ssl_context": {
        "check_hostname": false
      }
    }
  }
}
```

These are the keys in the configuration:

- *name* identifies a specific receiving system. It is a free-form string. Adapt to your needs - and remember it, we need it in the next step.
- *username* and *password* are the credentials that are needed on the receiving system. Use the *credentials you created when configuring the receiving system* (page 6).
- The systems address is specified using *url*.
- The users *mapping* inside *plugin_configs["kevlin"]* is as described above, only a bit longer.
- We also have a mapping for *school_classes*, which sets up the sync for those groups.
- *sync_password_hashes* - if password hashed should be synced.
- *ssl_context* - contains values that are passed to the `ssl.SSLContext`[19] object which is used to communicate with the receiving system.
- *active* - configures if this configuration for a an out queue for a school authority is active (so you don't have to delete it).
- *plugins* - which plugins are going to be used for this school authority. Usually just "kelvin".

Please adapt this to your needs, of course. The complete and adapted configuration needs to be posted to the `school_authorities` resource in the *Swagger UI* (page 8).

[19] https://docs.python.org/3.8/library/ssl.html#ssl.SSLContext

School to authority mapping

This is the second of the two *mappings* (page 8) we need.

While the above mapping defines which school authorities we have, we now need to map which school we sync to which authority - an authority could handle more than one school, so it's an 1:n mapping.

The format is:

```
{
  "mapping": {
    "NAME_OF_SCHOOL": "NAME_OF_RECIPIENT",
    "ANOTHER_SCHOOL": "OTHER_OR_SAME_RECIPIENT",
    ...
   }
}
```

You can have one or more schools in the mapping.

So assuming you have a `DEMOSCHOOL` on your sending system, and you used the above configuration to define `Traeger1` as a recipient system, you could do:

```
{
  "mapping": {
    "DEMOSCHOOL": "Traeger1"
   }
}
```

Note: *Remember?* (page 1) `Traeger` refers to the receiving side of the sync process

You can also find this example in *School to authority mapping example* (page 34).

Please "PUT" this configuration JSON to the `school_to_authority_mapping` resource in the *Swagger UI* (page 8).

Role specific attribute mapping

Note: This is an advanced scenario. If you don't need this, jump to the *next section* (page 13).

Back to our *example about telephone numbers* (page 8). Imagine that while telephone numbers should not be transferred for students, they are actually needed for teachers. This means, that we have a need to define per role which properties should be transferred.

With version `2.1.0` role specific attribute mapping was added to the default Kelvin plugin. This allows to define additional user mappings for each role (`student`, `teacher`, `staff` and `school_admin`) by adding a new mapping next to the `users` mapping suffixed by `_$ROLE`, e.g. `users_student: {}`.

If a user object is handled by the Kelvin plugin, the mapping is determined as follows:

1. Determine the schools the current school authority is configured to handle.
2. Determine all roles the user has in these schools.
3. Order the roles by priority: `school_admin` being the highest, followed by `staff`, `teacher` and then `student`.
4. Find a `users_$ROLE` mapping from the ones configured in the plugin settings, pick the one with the highest priority (from step 3).
5. If none was found, fall back to the `users` mapping as the default.

An example for such a configuration can be found in *Role specific Kelvin plugin mapping* (page 34)

Note: The priority order for the roles was chosen in order of common specificity in UCS@school. A student only ever has the role `student`. But teachers, staff and school admins can have multiple roles.

Note: The mappings for the different roles are not additive because that approach would complicate the option to remove mappings from a specific role. Therefore, only *one* mapping is chosen by the rules just described.

Warning: Users have the field `school_classes`, which describes which school classes they belong to. You might want to prevent certain user roles from being added or removed to school classes. Please be aware that leaving out the `school_classes` from the mapping is not sufficient to achieve this: changing the school classes of a user does not only result in a user change event but also a school class change event, which needs to be handled separately. You therefore need to use a derivative of the Kelvin plugin, which is described in the next section.

Partial group sync mapping

Note: This is an advanced scenario. If you don't need this, jump to the *next section* (page 13).

Remember that in the last examples we had a property that we would send for some users, but not others, depending on their role? Turns out that we can have the same problem for groups.

Imagine that a school manages locally which teachers belong to which class. In the role specific mapping we would *not* sync the classes attribute `school_classes`, preventing overwriting the local managed settings (*see above* (page 12)). This is not enough though: we would also need to make sure that we don't sync the property `users` of groups which contains those teachers.

With version `2.1.0` a new derivative of the Kelvin plugin was added: `kelvin-partial-group-sync`. This plugin alters the handling of school class changes by allowing you to specify a list of roles that should be ignored when syncing groups. The following steps determine which members are sent to a school authority when a school class is added:

1. Add all users that are members of the school class locally (Normal Kelvin plugin behavior).
2. Remove all users that have a configured role to ignore in any school handled by the school authority configuration.
3. Get all members of the school class on the target system that have one of the configured roles and add them.
4. Get all members of the school class on the target system that are unknown to the ID Connector and add them.

This results in school classes having only members with roles not configured to ignore,

- members with roles to ignore that were added on the target system,
- any users added on the target system which are unknown to the ID Connector.

Warning: To achieve this behavior, several additional LDAP queries on the ID Connector and one additional request to the target system are necessary. This affects performance.

To activate this alternative behavior replace the `kelvin` plugin in a school authority configuration with `kelvin-partial-group-sync`. The configuration options are exactly the same as for the `kelvin` plugin, except for the addition of `school_classes_ignore_roles`, which holds the list of user roles to ignore for school class changes.

See *Partial group sync* (page 35) for an example configuration.

Warning: Please be aware that this plugin can only alter the handling of dedicated school class change events. Due to the technical situation, changing the members of a school class often results in two events, a school class change and a user change. To actually prevent users of certain roles being added to school classes at all, it is necessary to leave out the mapping of the users `school_class` field in the configuration as well - *see above* (page 12).

1.1.5 Trying it out

Time has come to try it out. What we want to do:

1. Create a test user
2. Import the test user on the sending side
3. and watch the user being synced to the receiving side.

The slow way would be to create a user[20] individually (and make sure to amend the required properties), or to use the UCS@school import. You can read all about importing users in *UCS@school - Handbuch zur CLI-Import Schnittstelle* [6] (German only).

We however do it the fast way, creating and importing the user in one step:

```
$ /usr/share/ucs-school-import/scripts/ucs-school-testuser-import \
  --students 1  --classes 1 DEMOSCHOOL
```

This will create a user within a class in the school `DEMOSCHOOL`.

Now watch the log file to see the sync action on the *sender system*:

```
$ tail -f /var/log/univention/ucsschool-id-connector/queues.log
```

In another terminal on the *receiving system* you can see the user being received by the Kelvin API:

```
$ tail -f /var/log/univention/ucsschool-kelvin-rest-api/http.log # kelvin log
```

You might need to wait a short moment before the queue picks up the new user. If everything went fine, you should see some messages in the kelvin log, and you can confirm that the user was created in either the Kelvin web interface at `https://FQDN/ucsschool/kelvin/v1/docs` or in the UMC.

These log files are also a good starting point for debugging in case something went wrong.

Hint: When debugging always make sure that the following is correct and matches:

1. school authority configuration on the sender system (including auth credentials)
2. school to authority mapping on the sender system
3. `mapped_udm_properties.json` on the receiving system has all extra attributes that are defined in the school authority mapping.

Good luck! :-)

[20] https://help.univention.com/t/how-a-ucs-school-user-should-look-like/15630#a-sample-command-9

1.1.6 Starting / Stopping services

Both services (*ID Connector Service* and ID Connector *HTTP API*) run in a Docker container. The container can be started/stopped by using the regular service facility of the host system:

```
$ univention-app start ucsschool-id-connector
$ univention-app status ucsschool-id-connector
$ univention-app stop ucsschool-id-connector
$ univention-app restart ucsschool-id-connector
```

To restart individual services, the init scripts *inside* the Docker container can be used. The `univention-app` program has a command that makes it easy to execute commands *inside* the Docker container:

```
# UCS@School ID Connector service
$ univention-app shell ucsschool-id-connector /etc/init.d/ucsschool-id-connector␣
↪restart

# UCS@School ID Connector HTTP API
$ univention-app shell ucsschool-id-connector /etc/init.d/ucsschool-id-connector-
↪rest-api start
```

1.1.7 Updates

Updates are installed in one of the two usual UCS ways. Either via UMC or on the command line:

```
$ univention-upgrade

# or just:

$ univention-app upgrade ucsschool-id-connector
```

1.1.8 Extra: setting up a second school authority

If we already have a school authority set up and want to set up a second one (by copying its configuration) we can do the following:

1. Make sure the new school authority server has the Kelvin app installed and running.
2. Retrieve the configuration for our old school authority.

   For this we open the HTTP-API Swagger UI (`https://FQDN/ucsschool-id-connector/api/v1/docs`) and authenticate ourselves. The button can be found in the top right corner of the page.

   Then we retrieve a list of the available school authorities by using the `GET /ucsschool-id-connector/api/v1/school_authorities` tab, by clicking on `Try it out` and `Execute`.

   In the response body, we get a JSON list of the school authorities that are currently configured. We need to copy the one we want to replicate and save it for later.
3. Under `POST /ucsschool-id-connector/api/v1/school_authorities` we can create the new school authority.

   Click *try it out* and insert the copied JSON object from before into the request body.

   Now we just have to alter the name, URL, and login credentials before executing the request.

   - The URL has to point to the new school authorities HTTP-API.
   - The name can be chosen at your leisure
   - The password is the authentication token of the school authorities HTTP-API (retrieved earlier).

The tab `PATCH /ucsschool-id-connector/api/v1/school_authorities/{name}` can be used to change an already existing configuration.

To retrieve a list of the extended attributes on the old school authority server, one can use:

```
$ udm settings/extended_attribute list
```

1.2 Development

1.2.1 Overview

This diagram shows - in a different style - the diagram at the top of the last chapter *Administration* (page 3):

- The *school management software* that runs on the state level, and exports user data in a file format, e.g. CSV
- *UCS@school import* which is a python script to import users into a *DC Primary UCS* System
- The *DC Primary UCS* system passes on the user/group data to the actual *ID Connector* running in a docker container
- The *ID Connector* finally writes user/group data to the school authorities

This, of course, is a simplification. It is on a container level (in the sense used by C4[22]).

Note: Arrows in these diagrams are in the direction of data flow. It should be apparent from the source and target nodes what the label on the arrow refers to.

Let's have a look at what you need to know before we dive any deeper.

1.2.2 Prerequisites

First, we assume that you are already familiar with the chapter *Administration* (page 3), and the prerequisites described therein.

You also need the following knowledge to follow this manual and to develop for ID Connector:

HTTP The foundation of data communication for the WWW. Our APIs use this.

You need to be able to:

- understand HTTP messages
- understand auth concepts
- understand error codes

→ https://developer.mozilla.org/en-US/docs/Web/HTTP

Python & *pytest* The great programming language and its testing module.

You need to be able to:

- code and debug python modules
- test your code, ideally using *pytest*

→ https://www.python.org → https://pytest.org

FastAPI The framework in which our HTTP APIs are developed.

You need to be able to:

[21] https://c4model.com/
[22] https://c4model.com/

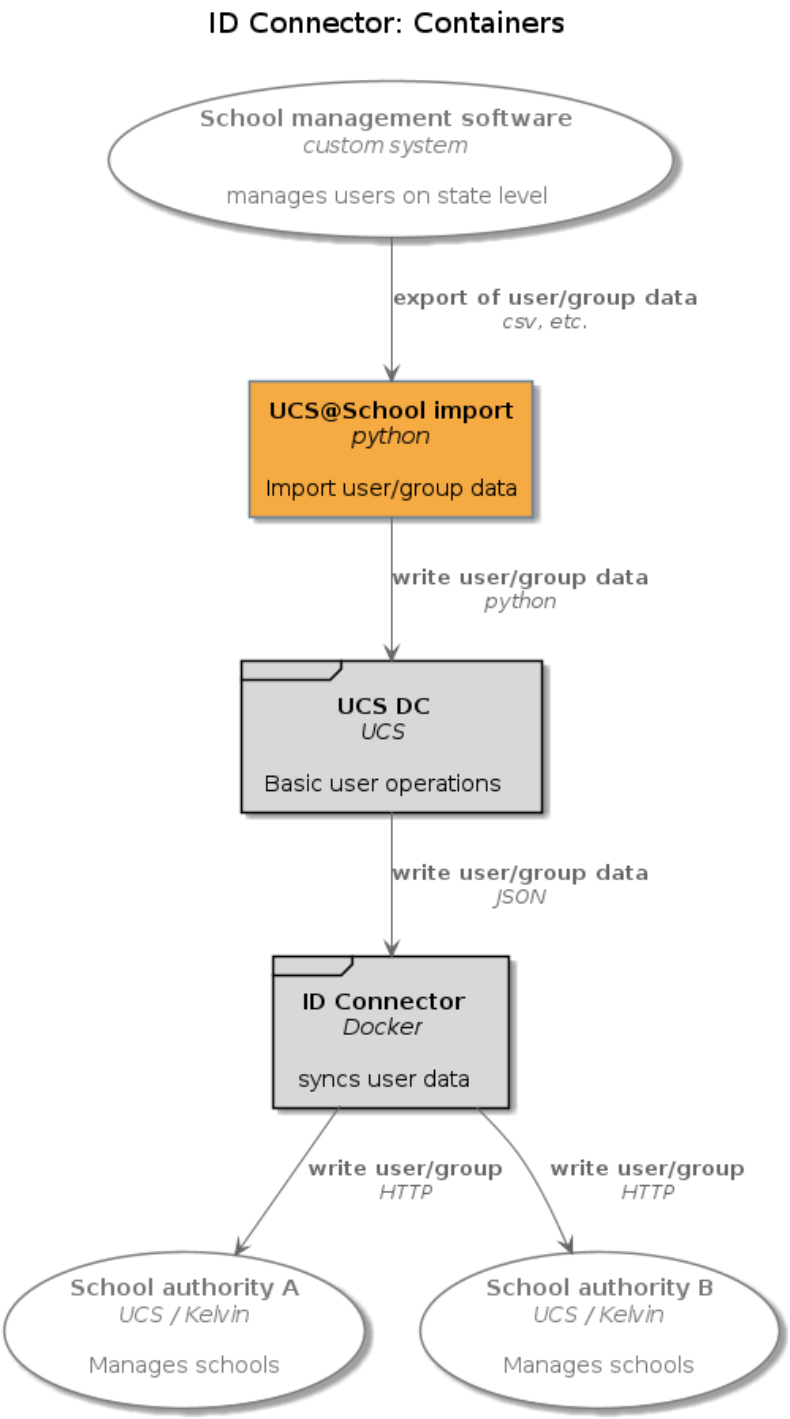


Fig. 2: ID Connector - Containers (C4 Style[21])

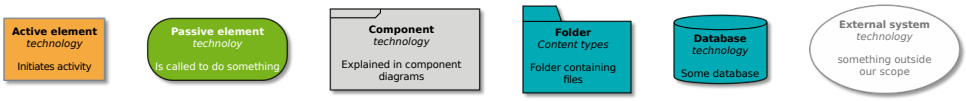


Fig. 3: Diagram elements

- understand *FastAPI*
- understand dependency injection
- understand *pydantic* models

→ https://fastapi.tiangolo.com/

Docker Software to isolate software and run them in containers.

You need to be able to:

- understand *Dockerfile* basics
- run containers
- understand mounts

→ https://www.docker.com/

Pluggy *Pluggy* is the crystallized core of plugin management and hook calling (from *pytest*).

You need to be able to:

- understand basic concepts of hook specs, hook implementation and hook calling

→ https://pluggy.readthedocs.io/en/latest/

UDM REST-API (optional) A REST API which can be used to inspect, modify, create and delete UDM objects via HTTP requests.

You will only need to know about this if you want to access extra information about objects within your (custom) plugin.

You need to be able to:

- understand the structure of UDM objects
- understand how to read (and maybe write) UDM objects, according to your needs

→ https://docs.software-univention.de/developer-reference/5.0/en/udm/rest-api.html

Pre-commit (optional) A framework for managing and maintaining multi-language pre-commit hooks.

This is only needed if you need to commit to the Univention ID Connector repository.

You need to be able to:

- install pre-commit definitions
- run pre-commit checks
- be aware of using different virtual environments for writing code and running pre-commits

→ https://pre-commit.com/

1.2.3 Interactions and components

Let's have a closer look at what's going on.

1.2.3.1 Overview, less simplified

OK, isn't this more or less the same as above in *Overview* (page 15)? Yes, right, you are. The additional element is the *Large in-queue*. This is a folder which interacts as the interface between the *DC Primary* and the *ID Connector*. JSON files are written to the folder, and then read out.

You also may notice the *get extra data* arrow. This means that the ID Connector might need extra data that is not contained in the JSON files. But before we come to this, we have a closer look at the *DC Primary*

1.2.3.2 DC Primary

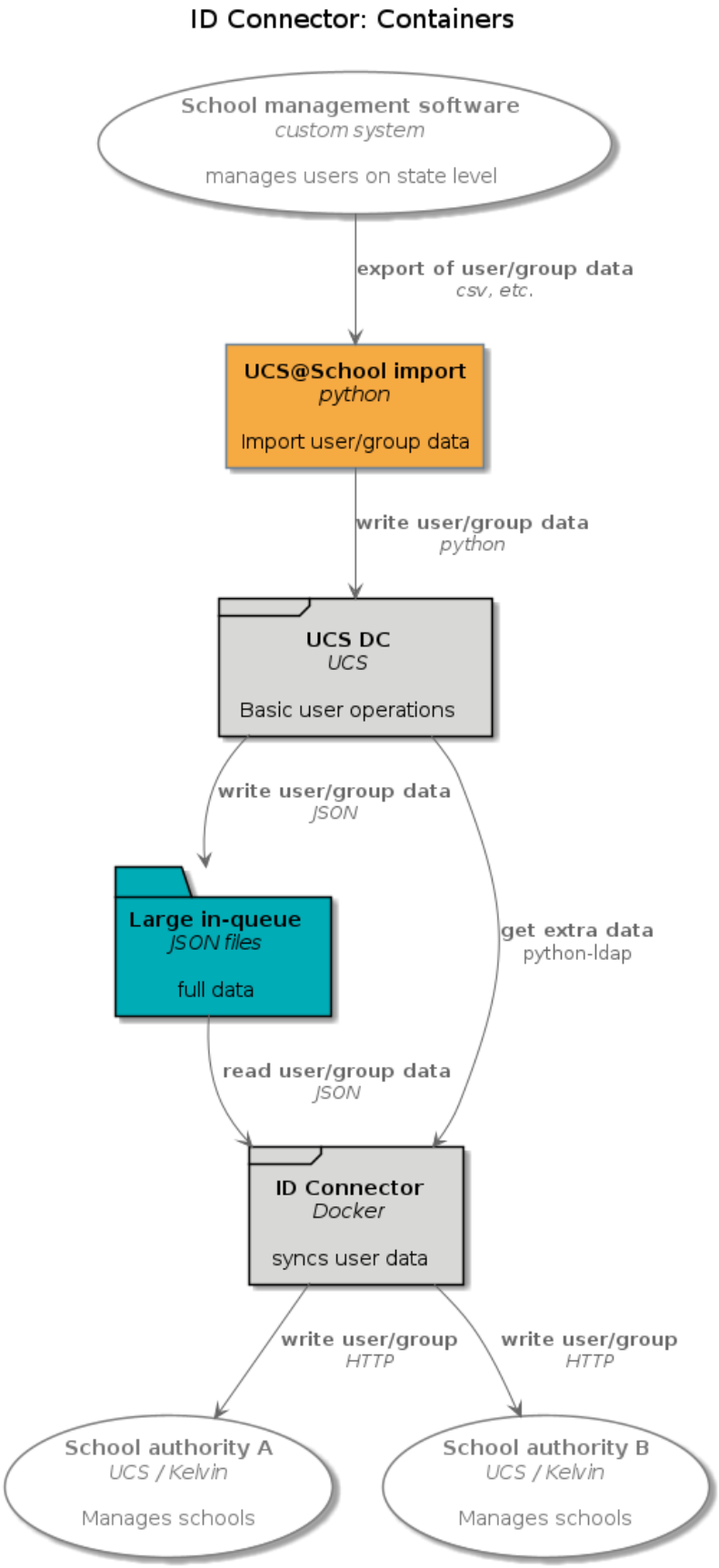
- The *Univention Corporate Server import*, a python script, reads in e.g. CSV data, and writes the contained user and group data to the *LDAP*. As mentioned in the diagram, there are other mechanisms that modify the LDAP, the UMC being one of them. The point is that user/group data "somehow" arrives.
- The *ID Connector listener* python script is then called by the LDAP machinery. It handles the write events that are of interest for the ID Connector.
- In a first step this data is written to the *small in-queue*. This is a folder containing minimal information (in JSON format), namely the type of change (add, update, delete) and the `entryUUID` of the concerned object. But why not write directly to the *Converter* in the next step? The reason is twofold:
  1. Speed by decoupling - the LDAP listeners should be able to do their job as fast as possible, and shouldn't have to wait for the next processing steps. Hence the folder acting as a queue, and only writing minimal data.
  2. The folder can also act as an entry point for debugging and manual insertion of user data. E.g. you want to reschedule a user without import the user again? Use the `schedule_user` script, and this will write some JSON into this folder.
- The *Converter* runs as a daemon script, picks up the JSON files from the *small in-queue*, and fetches the actual data from the *LDAP* using the `python-ldap`. It then puts a JSON representation of the UDM Object into the *Large in-queue*.

The *Large in-queue* in turn is read out by the *ID Connector* running in a docker container. Let's have a closer look:

1.2.3.3 ID Connector

- We already know that the *DC Primary* writes data to the *Large in-queue*. This folder is accessible by the host UCS system as well be the *ID Connector* docker container (where it is mounted).
- *In Queue* is a python process, that reads out the *Large in-queue*. It still might need extra data from the LDAP in the *DC Primary*, which it will do using `python-ldap`. For caching purposes it uses an `sqlite` database as a caching mechanism, the *UUID record cache*.
- The *In Queue* decides, what user/group data to send where (using the *School to authority mapping* (page 11) in the process). For each potential recipient there is a separate *Out queue*. User/group data is written in JSON format into the respective folder.
- The JSON data is picked up the plugin processes, e.g. *Out A*. Usually there is only the `Kelvin ID Connector plugin`, which helps ID Connector to talk to Kelvin REST APIs. The Kelvin plugin process then talks to Kelvin API on the *School authority A*, doing the final transmission of the user/group data.
- All this orchestrated by the *Management REST API*, which is mostly for managing out queues.

Hint: You have learned about *Management REST API* in *UCS@school ID Connector HTTP API* (page 8).


ID Connector: Containers
School management software
custom system
manages users on state level
export of user/group data
csv, etc.
UCS@School import
python
Import user/group data
write user/group data
python
UCS DC
UCS
Basic user operations
write user/group data
JSON
Large in-queue
JSON files
full data
get extra data
python-ldap
read user/group data
JSON
ID Connector
Docker
syncs user data
write user/group
HTTP
write user/group
HTTP
School authority A
UCS / Kelvin
Manages schools
School authority B
UCS / Kelvin
Manages schools

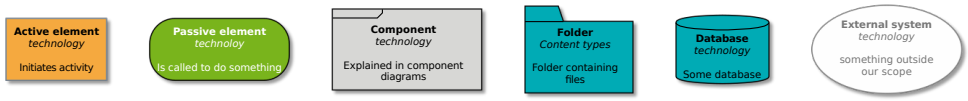


Fig. 4: Diagram elements

1.2.3.4 Complete picture

The complete picture is a bit crowded. If you want see it anyway, here are your choices:

1.2.4 Setup

Sadly, you can't develop everything on your developer machine / laptop: running the ID Connector requires an LDAP, listeners etc., so you really need a full-blown UCS installation. Hence, we rather have a local checkout on the development machine, and then sync the code changes into an ID Connector container that is running on a VM.

We are going to develop with the following setup:

- You have a git *checkout* of the *ucsschool-id-connector* on your *dev machine*
- There you use the script *devsync* to synchronize changes,
- which are synced to the corresponding *installation* folder of the *ID Connector* docker app.

Note: If you don't have *devsync* (from the *toolshed*), you might as well use **`scp`**, **`rsync`**, or any other transfer mechanism of your liking.

1.2.4.1 Machine

Setup development environment:

```
$ # clone ucsschool-id-connector
$ cd ucsschool-id-connector
$ make setup_devel_env
$ . venv/bin/activate
$ make install
$ pre-commit run -a
```

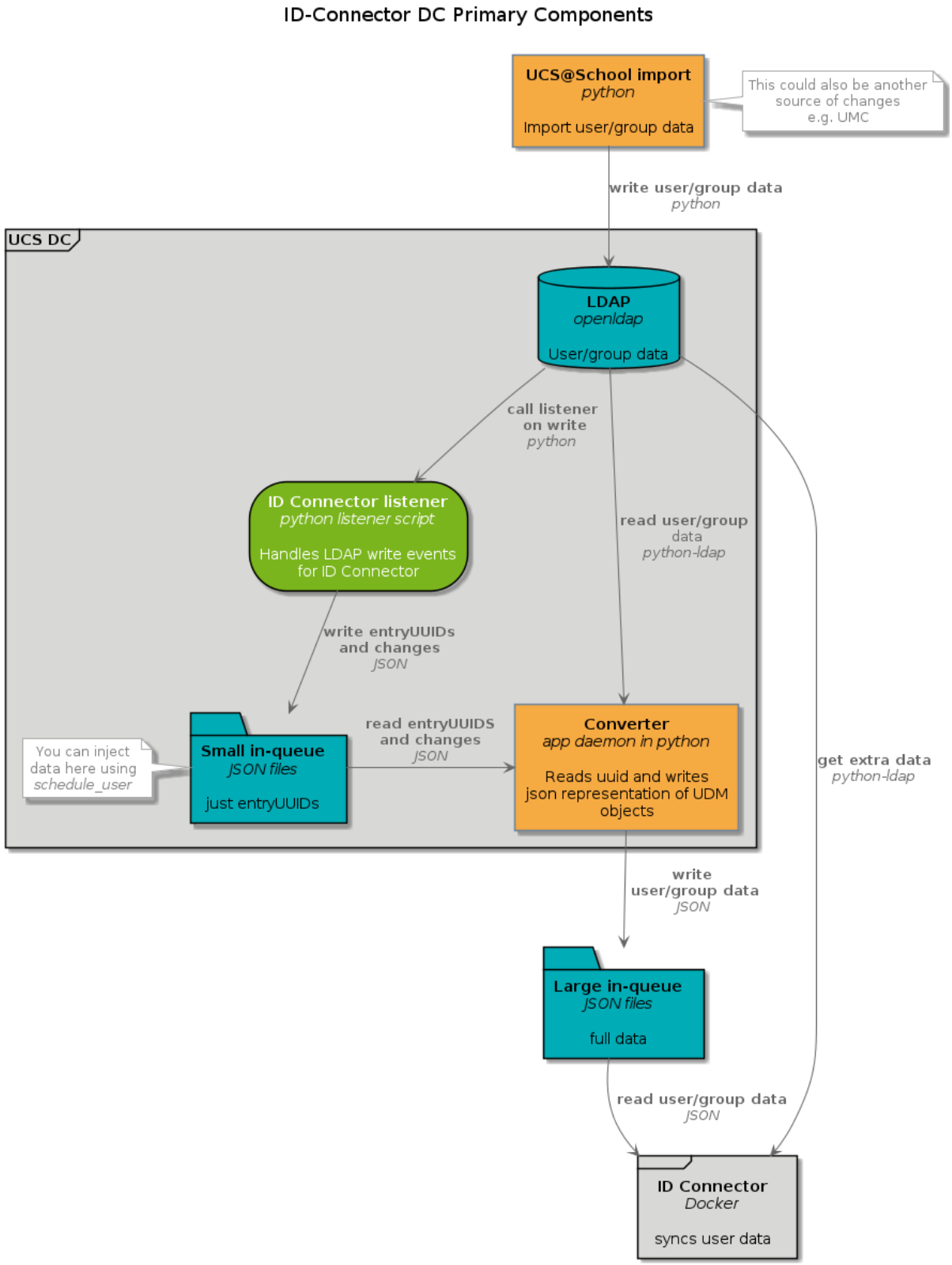
This will create a directory `venv` with a Python virtual environment with the app and all its dependencies in it.

You can later on also "activate" the `venv` using:

```
$ . venv/bin/activate
```

> **Warning:** All other commands in the `Makefile` assume that the *virtualenv* is activated.

Run `make` without argument to see more useful commands:


ID-Connector DC Primary Components
UCS@School import
python
Import user/group data
This could also be another source of changes e.g. UMC
write user/group data
python
UCS DC
LDAP
openldap
User/group data
call listener on write
python
ID Connector listener
python listener script
Handles LDAP write events for ID Connector
read user/group data
python-ldap
write entryUUIDs and changes
JSON
You can inject data here using schedule_user
Small in-queue
JSON files
just entryUUIDs
read entryUUIDS and changes
JSON
Converter
app daemon in python
Reads uuid and writes json representation of UDM objects
get extra data
python-ldap
write user/group data
JSON
Large in-queue
JSON files
full data
read user/group data
JSON
ID Connector
Docker
syncs user data

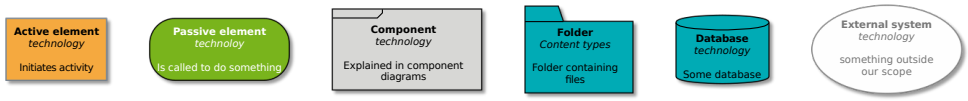


Fig. 5: Diagram elements

```
$ make

clean                     remove all build, test, coverage and Python artifacts
clean-build               remove build artifacts
clean-pyc                 remove Python file artifacts
clean-test                remove test and coverage artifacts
setup_devel_env           setup development environment (virtualenv)
lint                      check style (requires Python interpreter activated from␣
↪venv)
format                    format source code (requires Python interpreter␣
↪activated from venv)
test                      run tests with the Python interpreter from 'venv'
coverage                  check code coverage with the Python interpreter from
↪'venv'
coverage-html             generate HTML coverage report
install                   install the package to the active Python's site-packages
build-docker-img          build docker image locally quickly
build-docker-img-on-knut  copy source to docker.knut, build and push docker image
```

1.2.4.2 Virtual machine

You need to install the ID Connector app through the Univention App Center on your development VM.

When started through the Univention App Center use the following to enter the container of the app:

```
$ univention-app shell ucsschool-id-connector
```

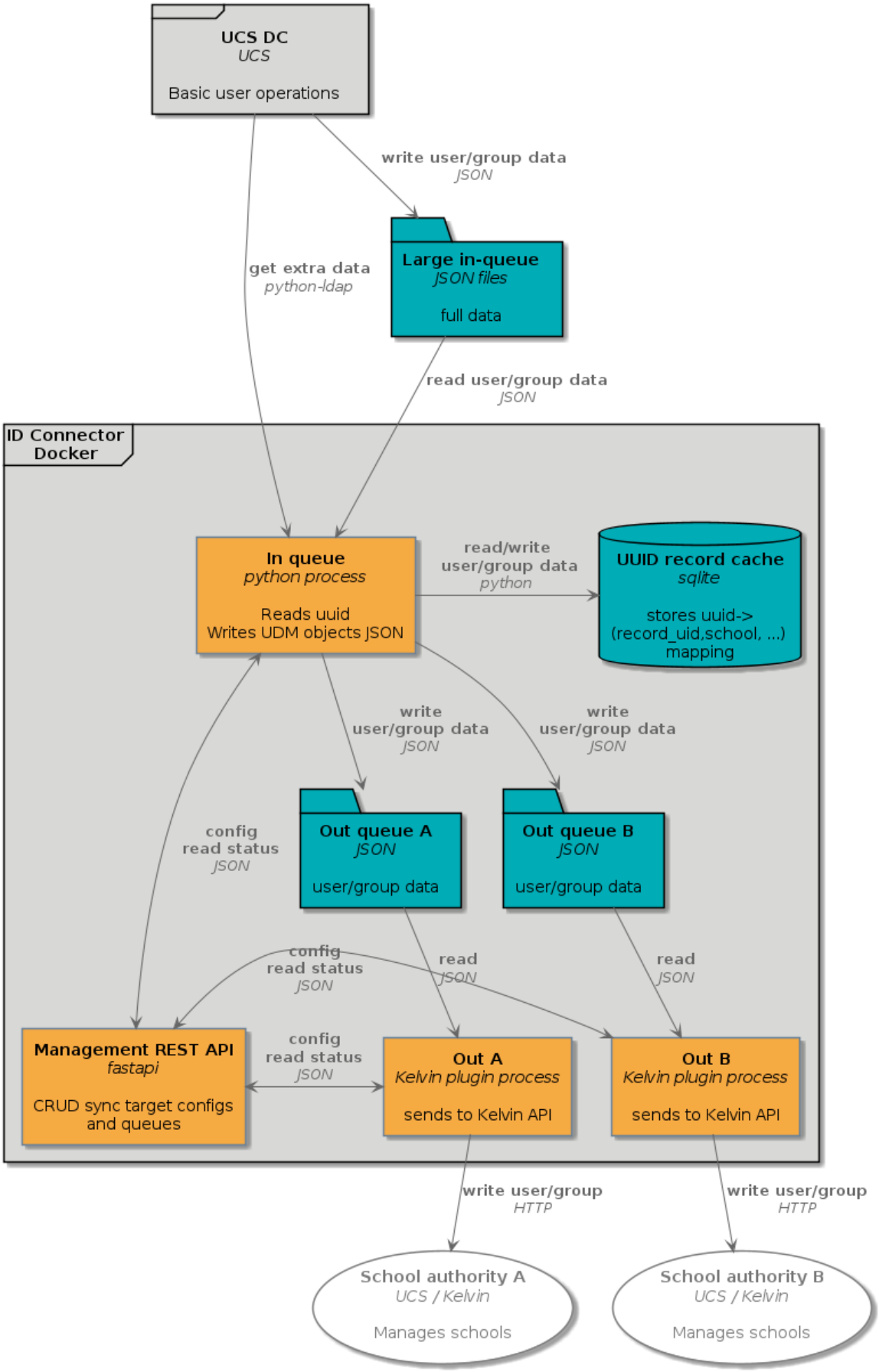
Inside the container, you can use the system Python:

```
/ucsschool-id-connector$ python3
Python 3.8.2 (default, Feb 29 2020, 17:03:31)
[GCC 9.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from ucsschool_id_connector import models

/ucsschool-id-connector$ ipython
Python 3.8.2 (default, Feb 29 2020, 17:03:31)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.13.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: from ucsschool_id_connector import models
```

Now, to synchronize your working copy into the running ID Connector container on the development virtual machine, we need to:

1. Stop the ID Connector in it's container,
2. Find out its ID,

ID Connector: ID Connector Components



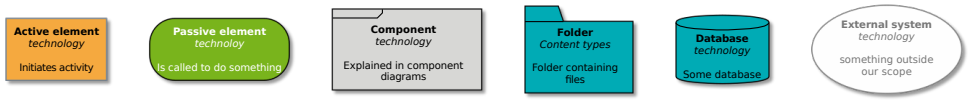
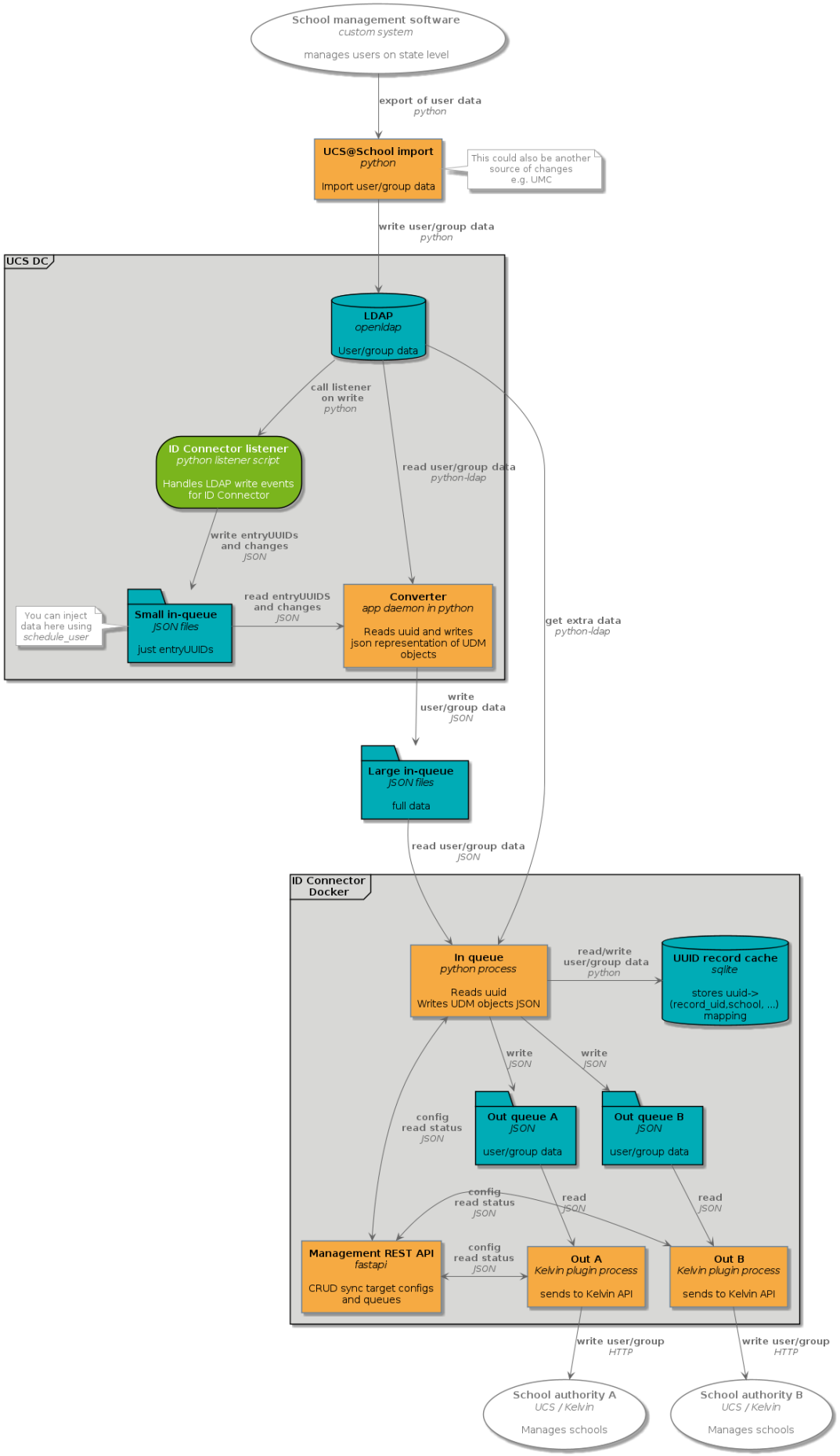


Fig. 6: Diagram elements

3. Use this ID to sync the files into the container,
4. Restart and prepare the container for development.

Lets do it!

```
# [dev VM] stop the actual ID Connector in it's docker container
$ docker exec "$(ucr get appcenter/apps/ucsschool-id-connector/container)" \
  /etc/init.d/ucsschool-id-connector stop

#[dev VM] find out the ID
$ docker inspect --format='{{.GraphDriver.Data.MergedDir}}' \
"$(ucr get appcenter/apps/ucsschool-id-connector/container)"

→  /var/lib/docker/overlay2/8dc...387/merged

# [developer machine] use the ID to sync our local modified files
ucsschool-id-connector$ devsync -v src/ \
<IP of dev VM>:/var/lib/docker/overlay2/8dc...387/merged/ucsschool-id-connector/

# [dev VM] enter the container
$ univention-app shell ucsschool-id-connector

# [in container] install the dev requirements
$ python3 -m pip install --no-cache-dir -r src/requirements.txt -r src/
↪requirements-dev.txt

# [in container] install ID Connector from source in development mode
$ python3 -m pip install -e src/

# [in container] start the ID Connector
$ /etc/init.d/ucsschool-id-connector restart

# [in container] stop the Rest API
$ /etc/init.d/ucsschool-id-connector-rest-api stop

# [in container] start the REST API in auto-reloading dev mode
$ /etc/init.d/ucsschool-id-connector-rest-api-dev start

# [in container] schedule a user
$ src/schedule_user demo_teacher

# DEBUG: Searching LDAP for user with username 'demo_teacher'...
# INFO : Adding user to in-queue: 'uid=demo_teacher,cn=lehrer,cn=users,
↪ou=DEMOSCHOOL,dc=uni,dc=dtr'.
# DEBUG: Done.

# Log is in /var/log/univention/ucsschool-id-connector/queues.log
```


School management software
custom system
manages users on state level
export of user data
python
UCS@School import
python
Import user/group data
This could also be another source of changes e.g. UMC
write user/group data
python
UCS DC
LDAP
openldap
User/group data
call listener on write
python
ID Connector listener
python listener script
Handles LDAP write events for ID Connector
read user/group data
python-ldap
write entryUUIDs and changes
JSON
You can inject data here using schedule_user
Small in-queue
JSON files
just entryUUIDs
read entryUUIDS and changes
JSON
Converter
app daemon in python
Reads uuid and writes json representation of UDM objects
get extra data
python-ldap
write user/group data
JSON
Large in-queue
JSON files
full data
read user/group data
JSON
ID Connector Docker
In queue
python process
Reads uuid
Writes UDM objects JSON
read/write user/group data
python
UUID record cache
sqlite
stores uuid-> (record_uid,school, ...) mapping
write
JSON
write
JSON
config read status
JSON
Out queue A
JSON
user/group data
Out queue B
JSON
user/group data
config read status
JSON
read
JSON
read
JSON
Management REST API
fastapi
CRUD sync target configs and queues
config read status
JSON
Out A
Kelvin plugin process
sends to Kelvin API
Out B
Kelvin plugin process
sends to Kelvin API
write user/group
HTTP
write user/group
HTTP
School authority A
UCS / Kelvin
Manages schools
School authority B
UCS / Kelvin
Manages schools

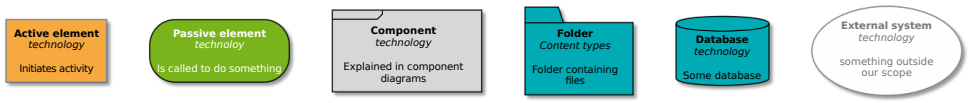


Fig. 7: Diagram elements

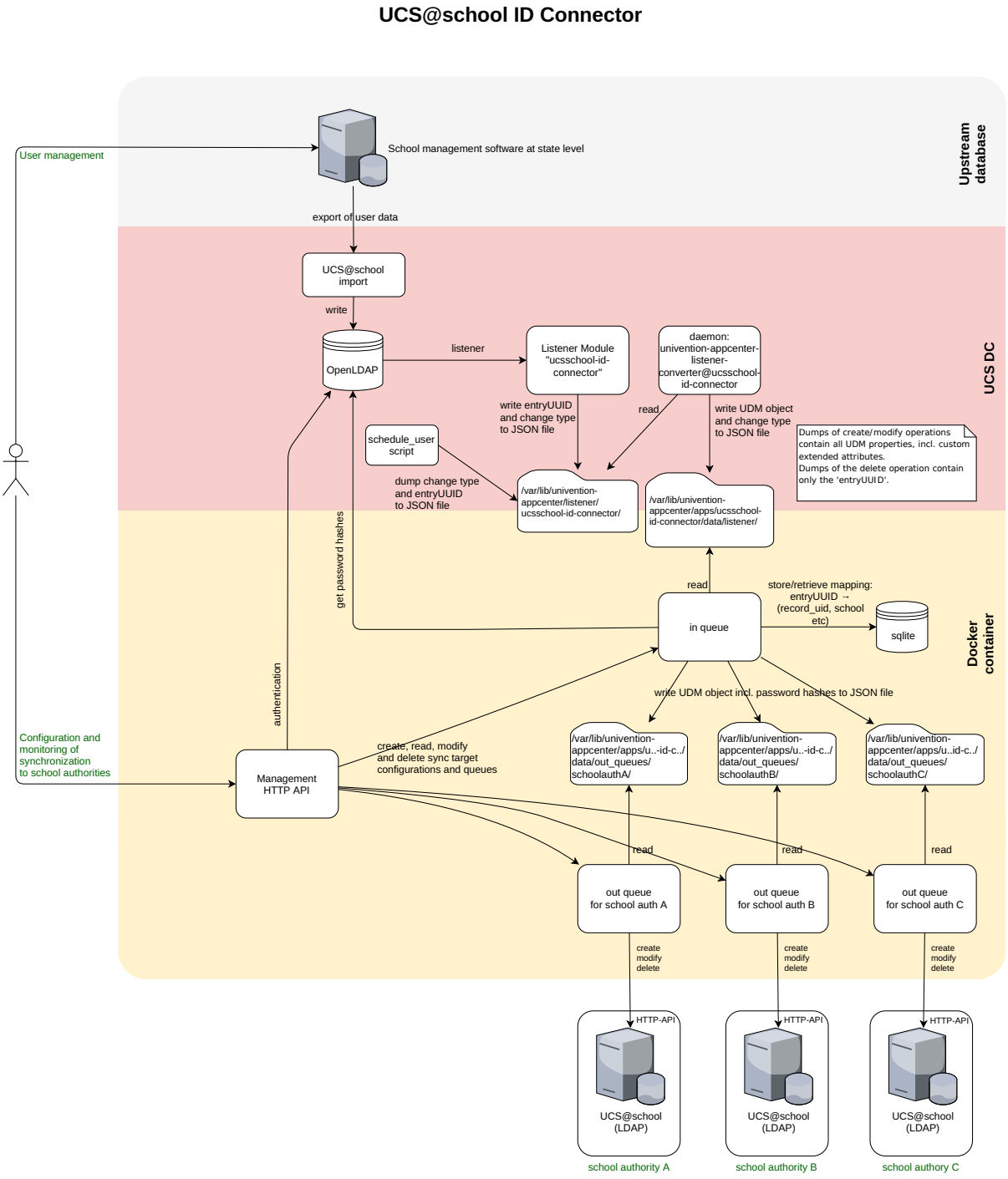


Fig. 8: The ID Connector, *not* simplified.

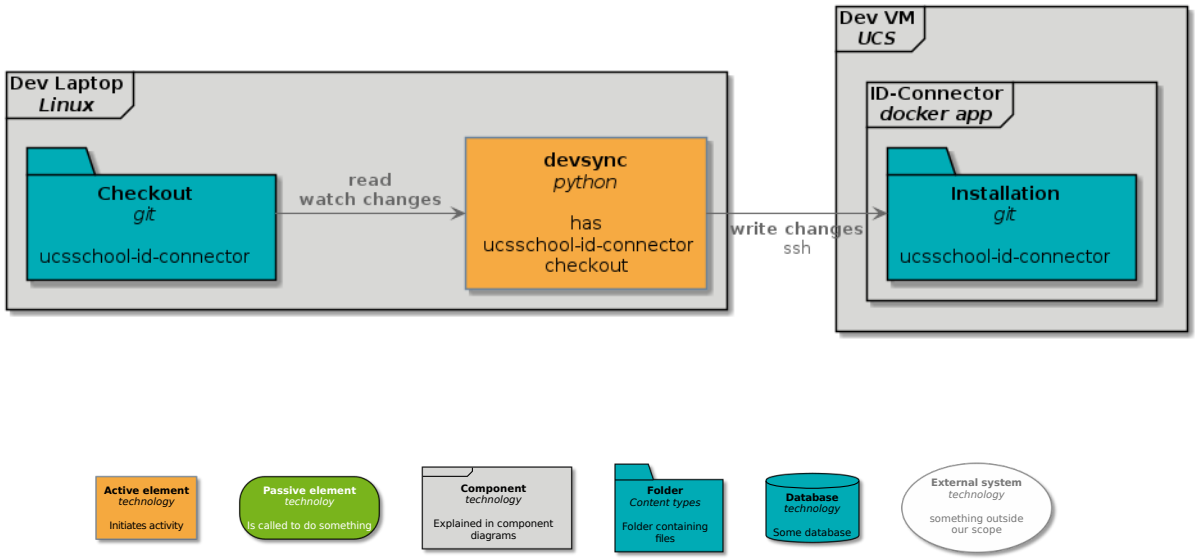


Fig. 9: Diagram elements

1.2.4.3 Run unit tests

Unit tests are executed as part of the *build process* (page 30). To start them manually in the installed apps running Docker container, run:

```
root@ucs-host:# univention-app shell ucsschool-id-connector
$ cd src/
$ python3 -m pytest -l -v tests/unittests
$ exit
```

1.2.5 Plugin development

1.2.5.1 How does the plugin system work?

The code of the *UCS@school ID Connector* app can be adapted through plugins. The pluggy[23] plugin system is used to define, implement and call plugins.

Hint: To get a quick freshen up on *Pluggy*, best have a look at the toy example[24] in the *Pluggy* documentation.

The basic idea:

- specify hook specifications: callables with the signature you want to have, decorated with a `hook_spec` marker.
- write actual hook implementations, a.k.a. 'plugins' that are later on called: callables with the same name and signature as in the specification, but this time decorated with a `hook_impl` marker

The `hook_spec` and `hook_impl` marker are already defined by the ID Connector system, you just need to use them. The same way, calling your custom plugin is also catered for, as well as finding your plugins.

The key file for ID Connector in this context is `src/ucsschool_id_connector/plugins.py`. In there you find the `hook_spec` and `hook_impl` markers.

[23] https://pluggy.readthedocs.io/en/latest/
[24] https://pluggy.readthedocs.io/en/latest/#a-toy-example

In this file you also find the plugin *specifications* (function signatures) - they are decorated with `@hook_spec`.

The app is released with default plugins, that implement a default version for all specifications found in `src/ucsschool_id_connector/plugins.py`. Search for `@hook_impl` in `src/plugins` to find them.

Some of the default plugins are only used if no custom plugins are present (see usages of `filter_plugins` defined in `src/src/ucsschool_id_connector/plugins.py`):

- create_request_keyword arguments (`kwargs`)
- school_authority_ping
- handle_listener_object

1.2.5.2 A simple custom plugin

The following demonstrates a simple example of a custom plugin for ID Connector.

The directory structure for your custom plugins (and packages, see below in *Advanced example* (page 29)) can be found in the host system in `/var/lib/univention-appcenter/apps/ucsschool-id-connector/conf/`:

```
/var/lib/univention-appcenter/apps/ucsschool-id-connector/conf/plugins/packages/
/var/lib/univention-appcenter/apps/ucsschool-id-connector/conf/plugins/plugins/
```

You can put a file containing a plugin class into the `plugins/plugins` directory. E.g. save the following into a file called `myplugin.py`:

```
from ucsschool_id_connector.utils import ConsoleAndFileLogging
from ucsschool_id_connector.plugins import hook_impl, plugin_manager
logger = ConsoleAndFileLogging.get_logger(__name__)

class MyPlugin:

    @hook_impl
    def get_listener_object(self, obj_dict):
        logger.info("Myplugin runs get_listener_obj with %r", obj_dict)

plugin_manager.register(MyPlugin())
```

Restart the ID Connector:

```
$ univention-app  restart ucsschool-id-connector
```

Now check the queues log in `/var/log/univention/ucsschool-id-connector/queues.log` and find entries like this:

```
2021-12-13 14:32:52 INFO  [ucsschool_id_connector.plugin_loader.load_plugins:79]␣
↪Loaded plugins:
[...]
2021-12-13 14:32:52 INFO  [ucsschool_id_connector.plugin_loader.load_plugins:81]  ␣
↪  'myplugin.MyPlugin': ['get_listener_object']
```

This tells you that `MyPlugin` was found and the hook implementation for `get_listener_object` was found.

1.2.5.3 Advanced example

In this example, you will learn how to additionally:

- define your own hook specs
- use an extra package for shared code across plugins.

The directory structure for a custom plugin `dummy` and custom package `example_package` below `/var/lib/univention-appcenter/apps/ucsschool-id-connector/conf/` looks as follows:

```
.../plugins/
.../plugins/packages
.../plugins/packages/example_package
.../plugins/packages/example_package/__init__.py
.../plugins/packages/example_package/example_module.py
.../plugins/plugins
.../plugins/plugins/dummy.py
```

Note: Putting the `example_package` into the `packages` directory solves an import problem. As the `packages` directory is appended to the `sys.path` by the module loader in `plugin_loader.py`, packages herein can be imported easily without being 'properly' installed.

Content of `plugins/packages/example_package/example_module.py`:

```
#
# An example Python module that will be loadable as "example_package.example_module
↪"
# if stored in 'plugins/packages/example_package/example_module.py'.
# Do not forget to create 'plugins/packages/example_package/__init__.py'.
#

from ucsschool_id_connector.utils import ConsoleAndFileLogging

logger = ConsoleAndFileLogging.get_logger(__name__)


class ExampleClass:
    def add(self, arg1, arg2):
        logger.info("Running ExampleClass.add() with arg1=%r arg2=%r.", arg1, arg2)
        return arg1 + arg2
```

Content of `plugins/plugins/dummy.py`:

```
#
# An example plugin that will be usable as "plugin_manager.hook.dummy_func()".
# It uses a class from a module in a custom package:
# plugins/packages/example_package/example_module.py
#

from ucsschool_id_connector.utils import ConsoleAndFileLogging
from ucsschool_id_connector.plugins import hook_impl, hook_spec, plugin_manager
from example_package.example_module import ExampleClass

logger = ConsoleAndFileLogging.get_logger(__name__)

class DummyPluginSpec:
    @hook_spec(firstresult=True)
    def dummy_func(self, arg1, arg2):
        """An example hook."""
```

(continues on next page)

(continued from previous page)

```
class DummyPlugin:
    @hook_impl
    def dummy_func(self, arg1, arg2):  # <-- this must match the specification!
        """
        Example plugin function.

        Returns the sum of its arguments.
        Uses a class from a custom package.
        """
        logger.info("Running DummyPlugin.dummy_func() with arg1=%r arg2=%r.", arg1,
↪ arg2)
        example_obj = ExampleClass()
        res = example_obj.add(arg1, arg2)
        assert res == arg1 + arg2
        return res


# register plugins
plugin_manager.register(DummyPlugin())
```

When the app starts, all plugins will be discovered and logged:

You will then find successful messages like this in the queues log in `/var/log/univention/ucsschool-id-connector/queues.log`:

```
...
INFO  [ucsschool_id_connector.plugins.load_plugins:83] Loaded plugins: {.., <dummy.
↪DummyPlugin object at 0x7fa5284a9240>}
INFO  [ucsschool_id_connector.plugins.load_plugins:84] Installed hooks: [..,
↪'dummy_func']
...
```

1.2.6 Building

1.2.6.1 Build docker image

Build the docker image:

```
$ make build-docker-img
```

The image can't easily be used productively, so this only for testing and development purposes:

```
$ docker run -p 127.0.0.1:8911:8911/tcp --name ucsschool_id_connector \
  docker-test-upload.software-univention.de/ucsschool-id-connector:$(cat VERSION.
↪txt)
```

Note: When the container is started that way (not through the Univention App Center) it must be accessed through `https://FQDN:8911/ucsschool-id-connector/api/v1/docs` after stopping the firewall (`service univention-firewall stop`).

You can also:

```
# let it run in the background.
$ docker run -d ...

# see the stdout
$ docker logs ucsschool_id_connector
```

(continues on next page)

(continued from previous page)

```
# stop the running container
$ docker stop ucsschool_id_connector

# remove the container
$ docker rm ucsschool_id_connector
```

To enter the running container run:

```
$ docker exec -it ucsschool_id_connector /bin/ash
```

1.2.6.2 Build release image

Warning: You need to be an Univention developer to use this section

- Update the apps version in `VERSION.txt`.
- Add an entry to `src/HISTORY.rst`.
- Build and push Docker image to Docker registry

To upload ("push") a new Docker image to the Univention Docker registry (`docker-test.software-univention.de`), run:

```
$ cd ~/git/ucsschool-id-connector
$ make build-docker-img-on-knut
```

1.2.7 Integration tests

Univention has automated integration tests. These are configured from this Jenkins configuration file:

https://github.com/univention/univention-corporate-server/blob/5.0-1/test/scenarios/autotest-244-ucsschool-id-sync.cfg

If you want to manually set up integration tests, for the moment, you need to look there for hints on how to do it.

1.3 File locations

This section lists relevant directories and files. Configuration file *must not* be edited by hand. All configuration is done either through the *app settings* in the UCS app center or through the *UCS@school ID Connector HTTP API*.

Nothing needs to be backed up and restored before and after an app update, because all important data is persisted in files on volumes mounted from the UCS host into the docker container.

1.3.1 Log files

`/var/log/univention/ucsschool-id-connector` is a volume mounted into the docker container, so it can be accessed from the host.

The directory contains:

- `http.log`: log of the HTTP-API (both ASGI server and API application)
- `queues.log`: log of the queue management daemon
- Old versions of above log files with timestamps appended to the file name. Log file rotation happens on Mondays and 15 copies are kept.

Log output can also be seen running:

```
$ docker logs <container name>
```

1.3.2 School authority configuration files

The configuration of the replication targets (*school authorities / Schulträger*) is stored in one JSON file per configured school authority under `/var/lib/univention-appcenter/apps/ucsschool-id-connector/conf/school_authorities`. The JSON configuration files must not be created by hand. The HTTP-API should be used for that instead.

Each school authority configuration has a queue associated.

1.3.3 Queue files

The LDAP listener process on the UCS host creates a JSON file for each creation/modification/move/deletion of a user object. Those JSON files are written to `/var/lib/univention-appcenter/apps/ucsschool-id-connector/data/listener`. That is the directory of the *in queue*.

The process handling the *in queue* copies files from there to a directory for each school authority that it can associate with the user account in the file. Each *out queue* handles a directory below `/var/lib/univention-appcenter/apps/ucsschool-id-connector/data/out_queues`.

When a school authority configuration is deleted, its associated queue directory is moved to `/var/lib/univention-appcenter/apps/ucsschool-id-connector/data/out_queues_trash`.

1.3.4 Token signature key

The key with which the JWTs are signed is in the file `/var/lib/univention-appcenter/apps/ucsschool-id-connector/conf/tokens.secret`. The file is created by the apps join script (see *Install* above).

1.3.5 SSL certificates for Kelvin client plugin

The plugin that connects to the Kelvin API on the school authority side looks for and stores SSL certificates as `/var/lib/univention-appcenter/apps/ucsschool-id-connector/conf/ssl_certs/HOSTNAME`. In case the certificate cannot be downloaded automatically, it can be saved there manually.

1.3.6 Volumes

The following directories are mounted from the host into the container:

- `/var/lib/univention-appcenter/listener`
- `/var/log/univention/ucsschool-id-connector`

1.4 Example json configurations

1.4.1 Sending system examples

1.4.1.1 School authority configuration

```
{
  "name": "Traeger1",
  "url": "https://10.200.3.242/ucsschool/kelvin/v1/",
  "active": true,
  "plugins": [
    "kelvin"
  ],
  "plugin_configs": {
    "kelvin": {
      "username": "Administrator",
      "password": "univention",
      "mapping": {
        "users": {
          "firstname": "firstname",
          "lastname": "lastname",
          "username": "name",
          "disabled": "disabled",
          "mailPrimaryAddress": "email",
          "e-mail": "email",
          "birthday": "birthday",
          "school": "school",
          "schools": "schools",
          "school_classes": "school_classes",
          "title": "title",
          "displayName": "displayName",
          "userexpiry": "expiration_date",
          "phone": "phone",
          "roles": "roles",
          "ucsschoolRecordUID": "record_uid",
          "ucsschoolSourceUID": "source_uid"
        },
        "school_classes": {
          "name": "name",
          "description": "description",
          "school": "school",
          "users": "users"
        }
      },
      "sync_password_hashes": true,
      "ssl_context": {
        "check_hostname": false
      }
    }
  }
}
```

1.4.1.2 School to authority mapping example

```
{
  "mapping": {
    "DEMOSCHOOL": "Traeger1"
   }
}
```

1.4.1.3 Role specific Kelvin plugin mapping

```
{
  "name": "Traeger2",
  "url": "https://10.200.3.242/ucsschool/kelvin/v1/",
  "active": true,
  "plugins": ["kelvin"],
  "plugin_configs": {
    "kelvin": {
      "mapping": {
        "school_classes": {
          "name": "name",
          "description": "description",
          "school": "school",
          "users": "users"
        },
        "users": {
          "firstname": "firstname",
          "lastname": "lastname",
          "username": "name",
          "disabled": "disabled",
          "mailPrimaryAddress": "email",
          "e-mail": "email",
          "birthday": "birthday",
          "school": "school",
          "schools": "schools",
          "school_classes": "school_classes",
          "ucsschoolSourceUID": "source_uid",
          "roles": "roles",
          "title": "title",
          "displayName": "displayName",
          "userexpiry": "expiration_date",
          "phone": "phone",
          "ucsschoolRecordUID": "record_uid"
        },
        "users_teacher": {
          "firstname": "firstname",
          "lastname": "lastname",
          "username": "name",
          "disabled": "disabled",
          "mailPrimaryAddress": "email",
          "e-mail": "email",
          "birthday": "birthday",
          "school": "school",
          "schools": "schools",
          "ucsschoolSourceUID": "source_uid",
          "roles": "roles",
          "title": "title",
          "displayName": "displayName",
          "userexpiry": "expiration_date",
          "phone": "phone",
          "ucsschoolRecordUID": "record_uid"
```

(continues on next page)

(continued from previous page)

```
        }
      },
      "password": "univention",
      "sync_password_hashes": true,
      "ssl_context": {
        "check_hostname": false
      },
      "username": "Administrator"
    }
  }
}
```

1.4.1.4 Partial group sync

This uses the `kelvin-partial-group-sync` plugin instead of the `kelvin` plugin in the *Role specific Kelvin plugin mapping* (page 34).

```
{
  "name": "Traeger2",
  "url": "https://10.200.3.242/ucsschool/kelvin/v1/",
  "active": true,
  "plugins": ["kelvin-partial-group-sync"],
  "plugin_configs": {
    "kelvin-partial-group-sync": {
      "school_classes_ignore_roles": ["teacher"],
      "mapping": {
        "school_classes": {
          "name": "name",
          "description": "description",
          "school": "school",
          "users": "users"
        },
        "users": {
          "firstname": "firstname",
          "lastname": "lastname",
          "username": "name",
          "disabled": "disabled",
          "mailPrimaryAddress": "email",
          "e-mail": "email",
          "birthday": "birthday",
          "school": "school",
          "schools": "schools",
          "school_classes": "school_classes",
          "ucsschoolSourceUID": "source_uid",
          "roles": "roles",
          "title": "title",
          "displayName": "displayName",
          "userexpiry": "expiration_date",
          "phone": "phone",
          "ucsschoolRecordUID": "record_uid"
        },
        "users_teacher": {
          "firstname": "firstname",
          "lastname": "lastname",
          "username": "name",
          "disabled": "disabled",
          "mailPrimaryAddress": "email",
          "e-mail": "email",
          "birthday": "birthday",
```

(continues on next page)

(continued from previous page)

```
          "school": "school",
          "schools": "schools",
          "ucsschoolSourceUID": "source_uid",
          "roles": "roles",
          "title": "title",
          "displayName": "displayName",
          "userexpiry": "expiration_date",
          "phone": "phone",
          "ucsschoolRecordUID": "record_uid"
        }
      },
      "password": "univention",
      "sync_password_hashes": true,
      "ssl_context": {
        "check_hostname": false
      },
      "username": "Administrator"
    }
  }
}
```

1.4.2 Receiving system examples

1.4.2.1 Mapped UDM properties

```
{
    "user": ["title", "phone", "e-mail"],
    "school": ["description"]
}
```

1.5 Changelog

v2.2.4 (2022-08-25)

- Users with multiple schools are now updated correctly if the Kelvin REST API is installed in version `1.5.4` or above on the school authority side.
- The permissions of the school authority configuration files was fixed.
- Kelvin REST API versions up to `1.7.0` are now supported. **Warning**: Kelvin REST API version `1.7.0` and above will break ID Connector versions below `2.2.4`.
- Remote school (OU) names are now compared case insensitively.

v2.2.2 (2022-03-03)

- The ID Broker plugin was removed from the app and can be installed separately by a Debian package.
- The ID Broker partial group sync plugin now safely handles group names with hyphen).
- Fixed users with multiple schools being created in alphabetical first, instead of same as in source domain.

v2.2.0 (2022-01-04)

- A new plugin was added to sync all user data to the ID Broker.
- The ID Connector can now also be installed on DC Backups.
- The Kelvin plugin can now be imported by other plugins, so they can subclass it.

- The synchronization of the `birthday` and `userexpiry` (in Kelvin `expiration_date`) attributes was fixed. The Kelvin REST API on the school authority side must be of version `1.5.1` or above!

v2.1.1 (2021-10-25)

- The log level for messages written to `/var/log/univention/ucsschool-id-connector/*.log` is now configurable. Valid values are `DEBUG`, `INFO`, `WARNING` and `ERROR`. Defaults to `INFO`.

v2.1.0 (2021-10-11)

- Update the integrated kelvin rest client to version `1.5.0` to work with Kelvin `1.5.0`
- Include kelvin plugin derivative for partial group sync

v2.0.1 (2021-03-04)

- The transfer of Kerberos key hashes has been fixed.

v2.0.0 (2020-11-10)

- Add Kelvin API plugin, which can be used with the ID Connector. The receiving side is required to have installed at least version `1.2.0` of the Kelvin API.
- The BB API plugin has been removed.

v1.1.0 (2020-06-02)

- The source code that is responsible for replicating users to specific target systems has been moved to plugins.
- The new variable `plugins` allows configuring which plugin to use for each school authority configuration.
- In combination the previous two features allow the connector to target a different API for each school authority.
- Update to Python 3.8.

v1.0.0 (2019-11-15)

- Initial release.

1.6 Bibliography

CHAPTER

TWO

INDICES AND TABLES

- genindex
- search

BIBLIOGRAPHY

[1] *UCS Manual*. Univention GmbH, 2021. URL: https://docs.software-univention.de/manual/5.0/en/.

[2] *Univention Developer Reference*. Univention GmbH, 2021. URL: https://docs.software-univention.de/developer-reference/5.0/en/.

[3] *Univention App Center for App Providers*. Univention GmbH, 2021. URL: https://docs.software-univention.de/app-center/5.0/en/.

[4] *UCS@school Kelvin REST API documentation*. Univention GmbH, 2021. URL: https://docs.software-univention.de/ucsschool-kelvin-rest-api.

[5] *UCS@school - Handbuch für Administratoren*. Univention GmbH, 2021. URL: https://docs.software-univention.de/ucsschool-manual/5.0/de/.

[6] *UCS@school - Handbuch zur CLI-Import Schnittstelle*. Univention GmbH, 2021. URL: https://docs.software-univention.de/ucsschool-import/5.0/de/.

INDEX

A

K

L

U