UCS@school ID Connector

Release 2.3.3

Univention GmbH

Jan 15, 2024

TABLE OF CONTENTS

License AGPL v3 [1] python 3.8 [2] Welcome to the documentation for the UCS@school ID Connector.

The ID Connector connects an UCS@school directory to any number of other UCS@school directories in a 1:n relation. It's designed to connect state directories with school districts, but you can also use it in other contexts. The connection takes place unidirectional. The connector transfers user data such as user, school affiliation, class affiliations from a central directory, for example a country directory, to district or school directories.

Prerequisite is the **`UCS@school`** *Kelvin* API in the school authority environments. This requires a configuration in advance to create an assignment *"Which remote instance needs which school users?"* Then the connector creates, updates, or deletes these users.

This documentation is for operators and system administrators who want to synchronize user identities between different school environments operated with UCS@school. You need to be familiar with the following topics:

- The concepts of UCS@school and Univention Corporate Server (UCS), such as the domain concept, UDM, and UCR.
- Software deployment on UCS, especially how to use the App Center and app settings
- UCS@school Kelvin REST API
- Work on the Linux command-line, view and edit text files, and examine log files.

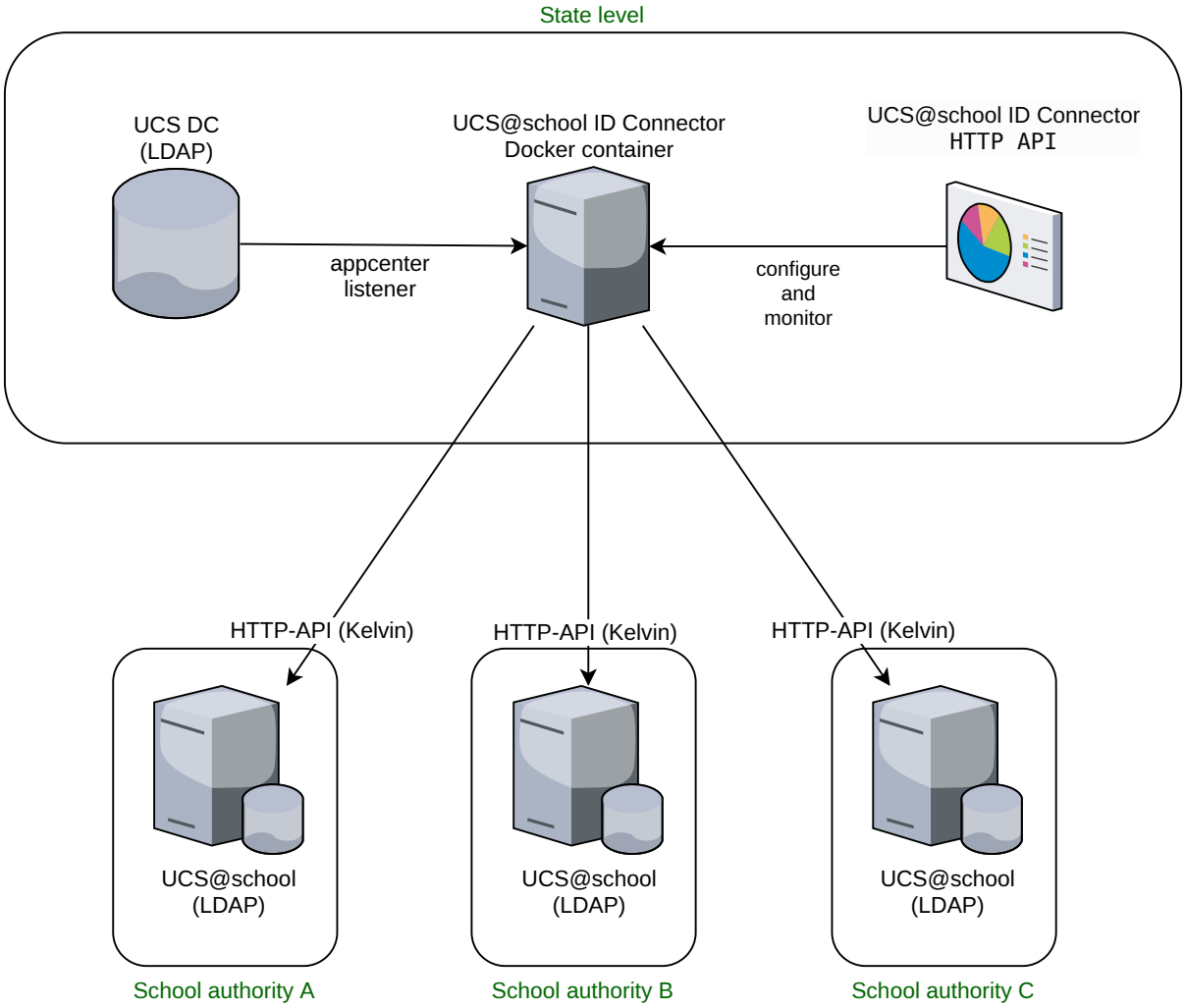

Fig. 1: Topology of the UCS@school ID Connector

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

In this documentation, you learn how to manage an ID Connector setup, and how to develop plugins for ID Connector.

Tip: The ID Connector setup is primarily used in German-speaking countries. Hence, you encounter a few German terms in this documentation.

Sender
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's the *recipient* of sync data.

Tip: You can use the clipboard icon on the top right of code examples to copy the code without Python and Bash prompts:

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

Hover with your mouse over the code to see the icon.

CHAPTER

ONE

ADMINISTRATION

This section describes the administration tasks for the ID Connector. It covers the topics installation, configuration of the sending system and the receiving systems and lifecycle tasks for the connector.

The ID Connector replication system consists of the following components as shown in Fig. 1.1:

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 through 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 Definitions

For the administration of and ID Connector setup or the integration with ID Connector, you need to make sure to know about the following aspects of a UCS environment.

LDAP and LDAP listener

UCS uses LDAP because of its optimization for reading in a hierarchical structure. Don't accessed directly, use *UDM* instead. OpenLDAP can have plugins, such as the notifier UCS heavily uses. 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. When notified, the listener calls listener modules, which are scripts in shell and Python.

You need to understand the basic concepts of LDAP.

See also:

For more information, see LDAP directory service[3] in *UCS Manual* [1].

Univention Directory Manager

UCS uses Univention Directory Manager (**UDM**) for handling user data and other data stored in the LDAP server. The LDAP server is one of two core storage locations. The other storage location is *UCR*. Examples for data are users, roles, or machine info.

UDM adds a layer of functionality and logic on top of LDAP, hence don't use LDAP directly, but only through UDM.

You need to:

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

[3] https://docs.software-univention.de/manual/5.0/en/index.html#introduction-ldap-directory-service

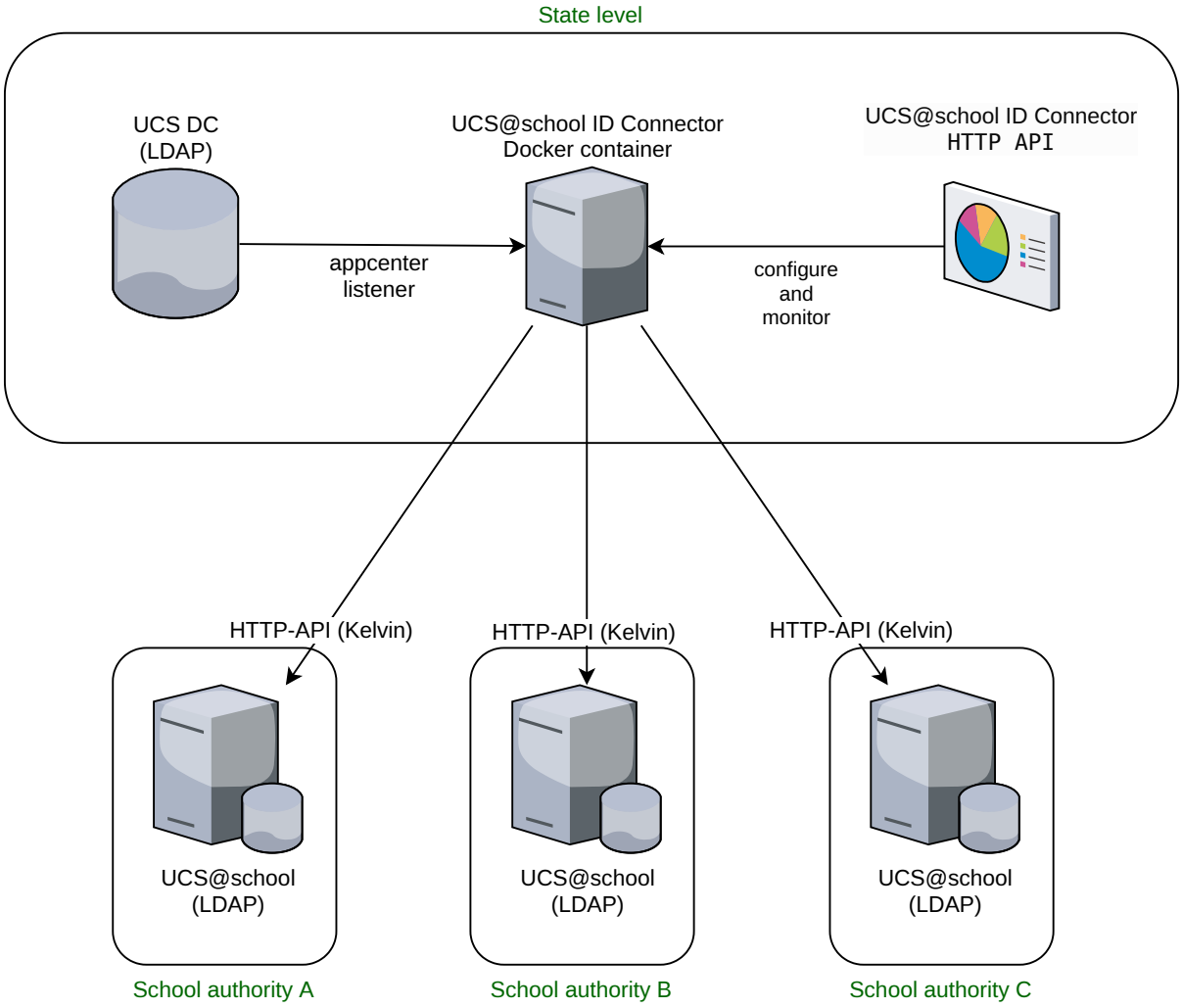

Fig. 1.1: Simplified overview of the ID Connector

See also:

For more information in a developer context, see Univention Directory Manager (UDM)[4] in *Univention Developer Reference* [2].

For an architecture overview, see Univention Directory Manager (UDM)[5] in *Univention Corporate Server 5.0 Architecture* [3].

Univention Configuration Registry

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

You need to:

- understand basic UCR concepts.
- know how to set and read UCR variables.

See also:

For more information, see Administration of local system configuration with Univention Configuration Registry[6] in *UCS Manual* [1].

For an architecture overview, see Univention Configuration Registry (UCR)[7] in *Univention Corporate Server 5.0 Architecture* [3].

Univention App Center settings

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

Within the App Center, you can configure settings for the individual apps.

See also:

For more information, see the following resources:

- App settings[8] in *Univention App Center for App Providers* [4]
- Setting of an application in the App Center[9] in *UCS Manual* [1]

UCS@school basics

Schools have special requirements for managing entities about what is going on inside them, such as teachers, students, staff, computer rooms, exams, and more. 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, and Kelvin is one of them.

You need to:

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

See also:

For more information, see the following resources:

- KB 15630 - How a UCS@school user should look like[10]
- KB 16925 - UCS@school work groups and school classes[11]
- UCS@school - Handbuch für Administratoren[12]

[4] https://docs.software-univention.de/developer-reference/5.0/en/udm/index.html#chap-udm
[5] https://docs.software-univention.de/architecture/5.0/en/services/udm.html#services-udm
[6] https://docs.software-univention.de/manual/5.0/en/computers/ucr.html#computers-administration-of-local-system-configuration-with-univention-configuration
[7] https://docs.software-univention.de/architecture/5.0/en/services/ucr.html#services-ucr
[8] https://docs.software-univention.de/app-center/5.0/en/configurations.html#app-settings
[9] https://docs.software-univention.de/manual/5.0/en/software/app-center.html#appcenter-configure
[10] https://help.univention.com/t/15630
[11] https://help.univention.com/t/16925
[12] https://docs.software-univention.de/ucsschool-manual/5.0/de/index.html

UCS@school Kelvin REST API

The UCS@school Kelvin REST API (Kelvin) provides HTTP endpoints to create and manage individual UCS@school domain objects such as school users, school classes and schools (OUs). It uses **`FastAPI`**, hence in **`Python 3`**.

You need to be able to install and configure Kelvin.

See also:

For more information, see the following resources:

- Overview[13] in *UCS@school Kelvin REST API documentation* [5]
- UCS@school-Objekte im LDAP-Verzeichnisdienst[14] in *UCS@school - Handbuch für Administratoren* [6]

If you want to also develop for the ID Connector, please also see the following section *Development* (page 21).

1.2 Installation

This section describes the installation of the UCS@school ID Connector app. For a working setup, you need a sending system with the ID Connector app, and receiving systems with the UCS@school Kelvin REST API.

1.2.1 Sending system

The UCS@school ID Connector app is available in the Univention App Center. You can install it with the following command:

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

The installation process runs the join script `50ucsschool-id-connector.inst` and creates the following:

- a key for signing the JWT tokens in the file `/var/lib/univention-appcenter/apps/ucsschool-id-connector/conf/tokens.secret`.
- the group `ucsschool-id-connector-admins` with the distinguished name (DN) `cn=ucsschool-id-connector-admins,cn=groups,$ldap_base`, whose members can access the ID Connector HTTP-API.

The section *Authentication* (page 9) explains both files in detail.

If the installation process didn't create the files listed before, you can re-run the join script with the following command.

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

Tip: You can validate the existence of the group with:

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

Tip: The installation process registers join scripts in the LDAP and runs them on any UCS system before, during, or after the join process.

For more information, see KB 13034 - A script shall be executed on each or a certain UCS systems before/during/after the join process[15]

[13] https://docs.software-univention.de/ucsschool-kelvin-rest-api/overview.html
[14] https://docs.software-univention.de/ucsschool-manual/5.0/de/structure.html#structure-ldap
[15] https://help.univention.com/t/13034

1.2.2 Receiving system

The UCS@school ID Connector app needs an HTTP-API on the target system to create, modify, and delete users. The ID Connector supports UCS@school Kelvin REST API.

The UCS@school ID Connector app synchronizes users for different domains between a sender and receiving systems. UCS takes care of the user synchronization within the domain.

To install the Kelvin API on each receiving system, run the following command:

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

To allow the UCS@school 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* [5].

1.3 Configure receiving system - HTTP-API (Kelvin)

You need to install and configure the Kelvin API as described in *UCS@school Kelvin REST API documentation* [5]. The following sections assume that you have installed the current version of Kelvin.

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

```
$ 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)"
```

Write down the credentials. You need them for the *school authority configuration on the sending system* (page 14).

Warning: You used the password before as input to a command. It's now in the command history. It's recommended to delete the command with the password from the command history. Run the following command:

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

After installation and basic configuration you can configure mapped UDM properties.

Beyond the standard object properties in UCS@school[16], you can define additional UDM properties and make them available in the Kelvin API on the target system.

1. To define additional UDM properties, you first create a mapping in the configuration file `/etc/ucsschool/kelvin/mapped_udm_properties.json`. The following example makes the listed properties additionally available for the resources `user` and `school`.

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

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

2. Restart Kelvin with the following command, for the configuration changes to take effect:

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

Caution: When configuring Kelvin in detail, 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.

Caution: Ensure that you configure all the mapped properties that the sending system sends, for example `displayName`. If the sender sends more than the receiver is configured to process, you experience unexpected errors, for example `404` in the log file.

1.4 Configure sending system

The school authorities configuration must be done through the ID Connector *HTTP API*. Don't edit configuration files directly.

1.4.1 UCS@school ID Connector HTTP API

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

`https://FQDN/ucsschool-id-connector/api/v1/queues/`
`/queues/` for the monitoring of queues

`https://FQDN/ucsschool-id-connector/api/v1/school_authorities/`
`/school_authorities/` for the configuration of school authorities

`https://FQDN/ucsschool-id-connector/api/v1/school_to_authority_mapping/`
`/school_to_authority_mapping/` for the configuration of which school you want to synchronize to which authority

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

- Swagger UI[17]: `https://FQDN/ucsschool-id-connector/api/v1/docs`
- ReDoc[18]: `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. You can download an OpenAPI v3 (formerly "Swagger") schema[19] from `https://FQDN/ucsschool-id-connector/api/v1/openapi.json`.

[17] https://github.com/swagger-api/swagger-ui
[18] https://github.com/Redocly/redoc
[19] https://swagger.io/docs/specification/about/

1.4.2 Authentication

Only users being member 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, you should only use the regular administration user accounts.

You can authorize yourself, for example in the Swagger UI using the *Authorize* button.

To use the ID Connector *HTTP API* from a script, the script must retrieve a JSON Web Token (JWT)[20] from `https://FQDN/ucsschool-id-connector/api/token`. The token is valid for a configurable amount of time, the default value for the time to life (TTL) is 60 minutes. To change the TTL of the token, open the corresponding *app settings* in the Univention App Center.

Listing 1.1: Example **`curl`** command to retrieve a token:

```
$ curl --include \
     --insecure \
     --request POST
     --data 'username=Administrator&password=s3cr3t' \
     https://FQDN/ucsschool-id-connector/api/token
```

1.4.3 Monitor processing status

This section describes how to monitor the processing status of the ID Connector queues. The status gives hints about how well the connector performs and if it works as intended.

When users and groups change in UCS, the ID Connector processes these changes as transactions and synchronizes them to the connected UCS@school domains. The connector puts the changes into queues to increase the robustness of the connector and to keep the load at a manageable level.

The ID Connector has an inbound queue that contains data coming from the App Center listener converter. The data uses a JSON representation of the changed objects. The ID Connector transforms the data from the inbound queue into transaction requests to the receiving systems. It buffers each transaction in an outbound queue. The ID Connector has one outbound queue per connected school authority (UCS@school domain) for outbound data.

Each queue is a directory and each transaction is a file in a queue directory. The queues locate at the following directories:

Inbound queue
`/var/lib/univention-appcenter/apps/ucsschool-id-connector/data/listener`

Outbound queues
`/var/lib/univention-appcenter/apps/ucsschool-id-connector/data/out_queues/`*`queue_name`*

The ID Connector provides the resources `/queues/` and `/queues/`*`name`*`/` that list the size of each queue. With the resource `/queues/`*`name`*`/` you can query a distinct queue and monitor it. To retrieve the size of a queue, use the following steps:

1. Authenticate yourself with the API as described in *Authentication* (page 9).
2. Request the data from the ID Connector API.

[20] https://en.wikipedia.org/wiki/JSON_Web_Token

Example

The following example shows how to query the API for all queue lengths. You can choose between a user interface approach with *Swagger UI*, or a command-line approach.

Use the following steps with the Swagger UI:

1. To authorize, click *Authorize* and enter the credentials of a legitimate user.
2. In the *queues* section, open the *GET* `/ucsschool-id-connector/api/v1/queues` resource and click *Try it out*.
3. Click the button *Execute*. In the *Server response* section, in the *Response body* area, you see a result similar to Listing 1.2.

Use the following commands for the command line:

1. Authorize yourself with the API and receive an `access_token`:

```
$ FQDN='<YOUR_FULLY_QUALIFIED_HOST_NAME>'
$ curl \
      --include \
      --insecure \
      --request POST \
      --data 'username=Administrator&password=s3cr3t' \
      https://"$FQDN"/ucsschool-id-connector/api/token
$ TOKEN='<YOUR_TOKEN>'
```

2. Request a list of queues and use the `access_token` you retrieved before:

```
$ curl \
      --insecure \
      --request 'GET' \
      'https://"$FQDN"/ucsschool-id-connector/api/v1/queues' \
      -H 'accept: application/json' \
      -H "Authorization: Bearer ${TOKEN}"
```

You see a result similar to Listing 1.2.

Hint: If you want to use a secure connection, you need download the UCS root certificate and pass it to **`curl`**:

```
$ F_PATH_CA_CERT="PATH_TO_UCS_ROOT_CERTIFICATE"
$ curl \
      --cacert "$F_PATH_CA_CERT" \
      --request 'GET' \
      'https://"$FQDN"/ucsschool-id-connector/api/v1/queues' \
      -H 'accept: application/json' \
      -H "Authorization: Bearer ${TOKEN}"
```

Listing 1.2: Example for a result

```
[
  {
    "name": "InQueue",
    "head": "",
    "length": 0,
    "school_authority": ""
  },
  {
    "name": "auth1",
    "head": "2024-01-11-13-43-36-196082_ready.json",
```

(continues on next page)

(continued from previous page)

```
    "length": 2,
    "school_authority": "auth1"
  },
  {
    "name": "auth2",
    "head": "",
    "length": 0,
    "school_authority": "auth2"
  }
]
```

Alerts for monitoring

If you want to add the UCS@school ID Connector to your monitoring environment and let the monitoring send you alerts, you may monitor the following problematic states.

Monotonous growth over a period of time
: If an ID Connector queue on the sending system grows continuously over a period of time, such as a day, the ID Connector isn't able to process transactions at the required speed as transactions arrive. Under normal circumstances, it may happen that the ID Connector can't process the transactions fast enough. If the queue sizes don't decrease at all for days, this could be a problem.

No change in queue size over a period of time
: If a queue size greater than `0` remains the same over a period of time, such as an hour, it indicates that the ID Connector isn't working or is stopping on corrupt transactions. If nothing changes in a queue and the size remains the same, you need to investigate. For more information, see *Interrupted processing* (page 11).

Queues don't reach a size of `0` for a period of time
: If the queues don't run empty over a period of time, such as a week, this can mean that transactions are coming in at the same rate as the connector is processing them. Or, the ID Connector is running too slowly overall. Or, the target system may be unreachable due to network problems or incorrect configuration.

Hint: The right period of time to trigger an alarm depends on your specific environment.

Interrupted processing

If the queue processing doesn't go as planned, for example, because the service is unavailable, the receiving system is unreachable, the ID Connector app has crashed, transactions are corrupt, or for any other reason, the queues grow in size or remain at a certain level. If the ID Connector app service doesn't run, the queues can quickly grow to a considerable size, such as more than 1 million files after some days.

If a transaction has a valid JSON format, but the receiver can't process it, the ID Connector moves the JSON file with the transaction from the queue to a trash directory for outgoing queues located at `/var/lib/univention-appcenter/apps/ucsschool-id-connector/data/out_queues_trash`.

If a transaction in JSON format located in any queue is corrupt, it may stay in the queue forever. To resolve an interrupted queue, use the following steps:

1. Find out, which transaction is corrupt.

   Have a look at the queue log file of the ID Connector on the sending system at `/var/log/univention/ucsschool-id-connector/queues.log`.

2. Resolve the error in the ID Connector setup.

   If other errors caused the corrupt transaction, you need to look at other parts. The ID Connector either logs a qualitative error happening on the sending system, or a generic error if a receiving system has an error with its Kelvin API. Review the configurations on the sending system and the receiving systems:

- On the sending system, validate the configurations of school authority objects in the UCS@school ID Connector. Have a close look at the URL, credentials, and attribute mappings.
- On the receiving systems, validate the attribute mapping and the incoming transactions through the Kelvin API. Have a look at the Kelvin log files[21].

For information about watching the transaction processing, see *Trying it out* (page 17).

3. Remove the corrupt transaction from the queue.

   Use the transaction you found in the log file, find the JSON file that contains the transaction, and remove the JSON file.

1.4.4 School authorities mapping

You need to configure the following things:

1. The school authorities you want to send data to, and what data can they receive. This section describes the procedure.
2. The actual schools the receiving system handles, meaning the school authority. The following section: *School to authority mapping* (page 14) describes the procedure.

Start with the first mapping, for school authorities.

To send user data to the target system, you must decide which properties of which objects the system needs to send, and more important, which properties **not** to send.

For example, there might be telephone numbers for students in the system on the sending side, but you don't want them available to the receiving school system. Instead of forbidding properties, you *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:

```
{
  "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:

- Synchronize the UDM `ucsschoolRecordUID` property to an UCS@school system as `record_uid`.
- Synchronize the UDM `ucsschoolSourceUID` property to an UCS@school system as `source_uid`.
- Synchronize the *virtual* `roles` property to an UCS@school system as `roles`.

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

Warning: When creating users through Kelvin, it requires some attributes. The following attributes must be present in the mapping:

[21] https://docs.software-univention.de/ucsschool-kelvin-rest-api/installation-configuration.html#file-locations

```
{
  "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 41).

Listing 1.3: Example of an ID Connector configuration for a school authority

```
{
  "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
      }
    }
  }
}
```

The mapping configuration uses the following keys:

name
identifies a specific receiving system. It's a free-form string. Adapt to your needs and remember it. You need it in the next step.

username and password
are the credentials that the receiving system needs. Use the *credentials you created when configuring the receiving system* (page 7).

url
specifies the address of the receiving system.

mapping
For a detailed description of the users' `mapping` inside `plugin_configs["kelvin"]`, see *School authorities mapping* (page 12).

school_classes
A mapping to setup the synchronization for school class groups.

sync_password_hashes
Set to `true`, if you want to synchronize the password hashes.

ssl_context
contains the values that the connector passes to the `ssl.SSLContext`[22] object. The connector uses this object to communicate with the receiving system.

active
set to `true` to activate the configuration for an out queue for a school authority. To deactivate the configuration, set the value to `false`.

plugins
Lists the plugins that the connector uses for this school authority. The list usually just has the element `"kelvin"`.

Adapt the configuration to your needs. You need to post the complete and adapted configuration to the `school_authorities` resource using the *Swagger UI* (page 8).

1.4.5 School to authority mapping

This section describes the second and last *mapping* (page 12) to setup the UCS@school ID Connector.

The before described mappings describe which school authorities you have. This section describes how to map which school you want to synchronize to which authority. A school authority can handle more than one school. You therefore have a `1:n` mapping.

The format for the mapping 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 configuration before to define `Traeger1` as a recipient system, you could do:

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

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

Note: *Remember?* (page 2) `Traeger` refers to the receiving side of the synchronization process.

You also find this example in *School to authority mapping* (page 14).

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

1.4.6 Role specific attribute mapping

Warning: This section describes an advanced scenario. If you don't explicitly need it, jump to the next section *Trying it out* (page 17).

Back to the *example about telephone numbers* (page 12). Imagine you don't want to transfer telephone numbers for students, you actually need them for teachers. This means, that you need to define per role which properties you want to transfer.

New in version 2.1.0: The default Kelvin plugin version `2.1.0` received the role specific attribute mapping feature. This allows to define additional user mappings for each role such as `student`, `teacher`, `staff` and `school_admin` by adding a mapping next to the `users` mapping suffixed by `_$ROLE`, for example `users_student: {}`.

If the Kelvin plugin handles a user object, the mapping looks like the following:

1. In the current school authority configuration, determine the schools that the authority handles.
2. Determine all roles the user has in these schools.
3. Order the roles by priority:
   - `school_admin` being the highest
   - `staff`
   - `teacher`
   - and then `student` with the lowest priority.
4. Find a `users_$ROLE` mapping from the ones configured in the plugin settings, pick the one with the highest priority.
5. If you couldn't fine any, fall back to the `users` mapping as the default.

You find an example for such a configuration in *Role specific Kelvin plugin mapping* (page 42).

Note: The priority order for the roles aligns with the order of common specificity in UCS@school.

A student only ever has the role `student`.

Teachers, staff, and school administrators can have multiple roles.

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

Warning: Users have the field `school_classes`, which describes which school classes they belong to. You can prevent certain user roles from adding to or removing from school classes.

Be aware that leaving out the `school_classes` from the mapping isn't sufficient to achieve this. Changing the school classes of a user doesn't only result in a user change event, but also a school class change event, which you need to handle separately.

You therefore need to use a derivative of the Kelvin plugin, as described in *Partial group sync mapping* (page 16).

1.4.7 Partial group sync mapping

Warning: This section describes an advanced scenario. If you don't explicitly need it, jump to the next section *Trying it out* (page 17).

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

Imagine that a school manages locally which teachers belong to which class. In the role specific mapping you would **not** synchronize the classes attribute `school_classes`, to prevent overwriting the local managed settings, *see above* (page 15).

This isn't enough though. You would also need to make sure that you don't synchronize the property `users` of groups which contains those teachers.

New in version 2.1.0: Kelvin plugin version `2.1.0` adds the derivative `kelvin-partial-group-sync`. This plugin alters the handling of school class changes by allowing you to specify a list of roles that you want to ignore when synchronizing groups.

The following steps determine which members the connector sends to a school authority, when an administrator adds a school class:

1. Add all users that are members of the school class locally. This is the default 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 that have 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 the `kelvin-partial-group-sync` plugin. 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.

For an example configuration, see *Partial group sync* (page 43).

Warning: 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
- a user change

To actually prevent the ID Connector to add users of certain roles to school classes at all, it's necessary to leave out the mapping of the users `school_class` field in the configuration as well - *see the previous section* (page 15).

1.5 Trying it out

Time has come to verify the setup. In this section, you go through the following steps:

1. Create a test user.
2. Import the test user on the sending side.
3. Watch the synchronization of the test user to the receiving side.

Caution: The ID Connector only synchronizes users with the properties `ucsschoolRecordUID` and `ucsschoolSourceUID`. You need to ensure that the user accounts used for testing have these properties.

The slow way is to create a user[23] individually and to ensure to amend the required properties, or to use the UCS@school import. For more information about user import in UCS@school, see *UCS@school - Handbuch zur CLI-Import Schnittstelle* [7], German only.

With the fast way, you create and import the user in one step. The following command creates a user within a class in the school `DEMOSCHOOL`.

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

To see the synchronization action on the *sender system*, you can watch the log file with the following command:

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

To see the synchronization action on the *receiving system*, you can watch the log file of Kelvin with the following command:

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

You may need to wait a short moment before the queue picks up the created user. If everything went fine, you see some messages in the Kelvin log file on the receiving system. You can confirm that the ID Connector created the user in either the Kelvin web interface at `https://FQDN/ucsschool/kelvin/v1/docs`, or in the UMC.

The following log files are also a good starting point for debugging in case something went wrong:

- On the sending system: `/var/log/univention/ucsschool-id-connector/queues.log`
- On the receiving system: `/var/log/univention/ucsschool-kelvin-rest-api/http.log`

Important: During debugging, you must always ensure that the following is correct and matches:

1. School authority configuration on the sender system, including authentication 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.

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

1.6 Starting / Stopping services

Both services, ID Connector *Service* and ID Connector *HTTP API*, run in a Docker container. You can start or stop the container by using the regular service facility of the UCS 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, you can use the init scripts *inside* the Docker container. **`univention-app`** has the `shell` command that allows to run 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
```

1.7 Updates

To install updates for ID Connector, use one of the two usual UCS ways. Either through UMC or on the command line:

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

1.8 Extra: setting up a second school authority

If you already have a school authority set up and you want to set up another one by copying its configuration, you can do the following:

1. Make sure the school authority server that you want to add has the Kelvin app installed and running.
2. Retrieve the configuration for the old school authority.

   For this you open the HTTP-API Swagger UI at `https://FQDN/ucsschool-id-connector/api/v1/docs` and *authenticate yourself* (page 9).

   To retrieve a list of the available school authorities, use the `GET /ucsschool-id-connector/api/v1/school_authorities` tab in the Swagger UI, click *Try it out* and *Execute*.

   In the response body, you receive a list of school authorities in JSON format, that the ID Connector has configured. You need to copy the school authority that you want to replicate and save it for later.
3. At the tab `POST /ucsschool-id-connector/api/v1/school_authorities` in the Swagger UI you can create the school authority.

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

   Before you execute the request, you must alter the name, URL, and login credentials:

   - The URL must point to the school authority's Kelvin HTTP-API.
   - You can choose the name at your leisure.

- The password is the authentication token of the school authority's HTTP-API that you retrieved earlier.

Use the tab `PATCH /ucsschool-id-connector/api/v1/school_authorities/name` to change an already existing configuration.

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

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

CHAPTER

TWO

DEVELOPMENT

This section is for software developers who want to setup an environment to develop UCS@school ID Connector. It provides an overview of the architecture, the components and their interactions.

Fig. 2.1 shows the C4 style of Fig. 1.1 from the last chapter *Administration* (page 3):

- The *school management software* that runs on the state level, and exports user data in a file format, for example 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 and group data to the actual **`ID Connector`** running in a Docker container.
- The **`ID Connector`** finally writes user and group data to the school authorities.

Fig. 2.3 is a simplification. It's on a container level, in the sense used by the C4 model[25].

Note: Arrows in C4 model 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.

2.1 Development prerequisites

This section assumes that you are already familiar with the section *Administration* (page 3), and the *Definitions* (page 3) 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 internet. The ID Connector APIs use HTTP. You need to understand:

- HTTP messages
- authentication concepts
- error codes

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

Python and `pytest`

The Python programming language and its testing module. You need to:

- program and debug Python modules
- test Python source code, ideally using **`pytest`**

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

[24] https://c4model.com/
[25] https://c4model.com/

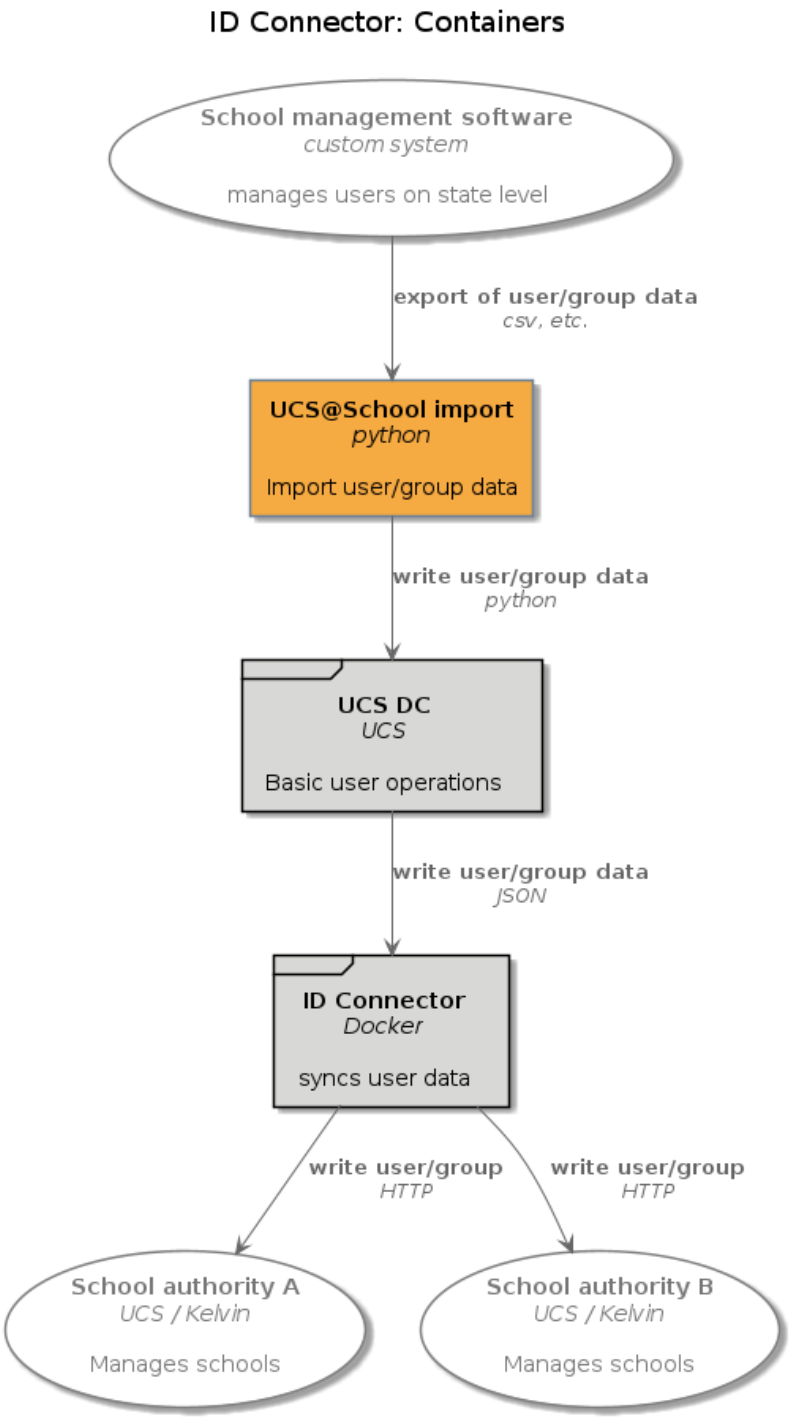

Fig. 2.1: ID Connector - Containers (C4 Style[24])

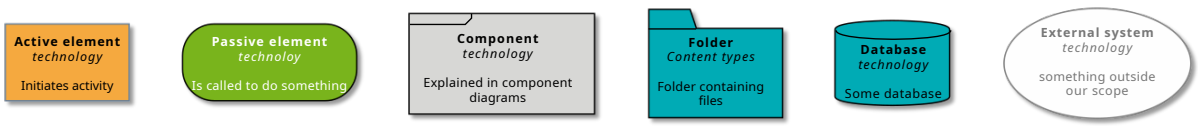

Fig. 2.2: Diagram elements

`FastAPI`

ID Connector uses the **`FastAPI`** framework for the APIs. You need to understand:

- **`FastAPI`**
- dependency injection
- *Pydantic* models

→ https://fastapi.tiangolo.com/ → https://docs.pydantic.dev/latest/

Docker

Software to isolate software and run them in containers. You need to:

- understand `Dockerfile` basics
- run containers
- understand mounts

→ https://www.docker.com/

`Pluggy`

`Pluggy` is the **`pytest`** plugin system. You need to understand basic concepts of hook specifications, hook implementation and hook calling.

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

UDM REST API (optional)

A HTTP REST API which you can use to inspect, modify, create, and delete UDM objects through HTTP requests.

You only need to know about the UDM REST API if you want to access extra information about objects within your custom plugin. You need to understand:

- the structure of UDM objects.
- 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. You only need **`pre-commit`**, if you commit to the Univention ID Connector repository. You need to understand:

- how to install **`pre-commit`** definitions.
- how to run **`pre-commit`** checks.
- be aware of using different virtual environments for writing code and running **`pre-commit`**.

→ https://pre-commit.com/

2.2 Interactions and components

This section describes the interactions between the components.

2.2.1 Overview, less simplified

Fig. 2.3 shows the containers for the ID Connector.

Compared to Fig. 2.1, the additional element is *Large in-queue* on the left in the middle. It's a folder which interacts as the interface between the *Primary Directory Node* and the **`ID Connector`**. JSON files are written to the folder, and then read out.

The *get extra data* arrow is an interaction from the ID Connector when it might need extra data that isn't contained in the JSON files.

2.2.2 Primary Directory Node

Fig. 2.5 gives a detailed view on the *UCS DC* component located between *UCS@school import* and *Large in-queue*. This section describes the elements in detail.

The *Univention Corporate Server import* is a Python script, that reads data such as 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 and group data *somehow* arrives.

The LDAP machinery then calls the **`ID Connector`** *ID Connector listener* Python script is. The *ID Connector listener* handles the write events that are of interest for the ID Connector.

In a first step, the *ID Connector listener* writes this data to the *small in-queue*, a folder containing minimal information in JSON format, namely the type of change, such as add, update, delete, and the `entryUUID` of the concerned object.

The *ID Connector listener* doesn't write the data directly to the *Converter* for the following reasons:

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 acts as a queue, and only writes minimal data.
2. The folder can also act as an entry point for debugging and manual insertion of user data. For example, you want to reschedule a user without import the user again.
   - To write some JSON data into this folder, use the **`schedule_user`** script.
   - Or for groups, use the **`schedule_group`** script.
   - To add JSON data into the folder for all school users and school groups, use the **`schedule_school`** script.

The *Converter* runs as a daemon, 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*.

In turn, the **`ID Connector`** running in a Docker container reads the *Large in-queue*.

2.2.3 ID Connector

Fig. 2.7 shows the *ID Connector* component between *Large in-queue* and the *School authority*. This section describes the elements in detail.

As described in *Primary Directory Node* (page 24), the *Primary Directory Node* writes data to the *Large in-queue*. The host UCS system and the **`ID Connector`** Docker container can access the *Large in-queue* folder. The Docker container actually mounts the folder.

In Queue is a Python process, that reads the *Large in-queue*. It may need extra data from the LDAP on the *Primary Directory Node*, which it fetches using `python-ldap`. For caching purposes, it uses an **`SQLite`** database as a caching mechanism, called the *UUID record cache*.

The *In Queue* decides, what user and group data to send where. It uses the *School to authority mapping example* (page 42) for decision in the process. For each potential recipient there is a separate *Out queue*. It writes user and group data in JSON format into the respective folder.

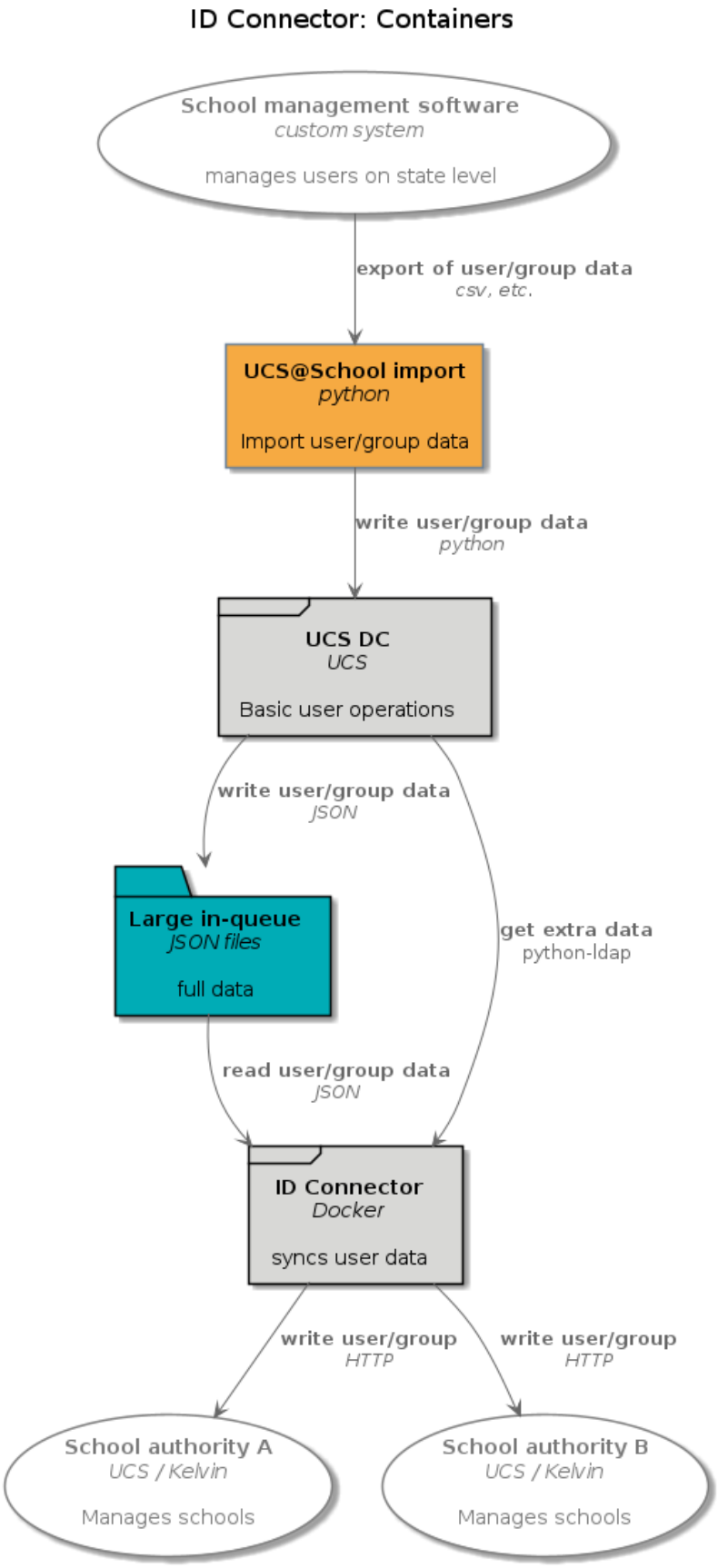

Fig. 2.3: ID Connector containers

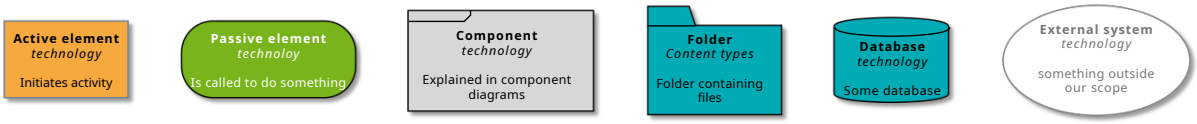

Fig. 2.4: Diagram elements

The plugin processes pick up the JSON data, for example *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 and group data.

The *Management REST API* orchestrates the processes and manages the outgoing queues.

Hint: To read more about *Management REST API*, see *UCS@school ID Connector HTTP API* (page 8).

2.2.4 Complete picture

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

2.3 Setup for development

This section describes the setup of ID Connector for development.

Running the ID Connector requires an LDAP, listeners etc., so you really need a complete UCS installation. Hence, you rather have a local checkout on the development machine, and then synchronize the code changes into an ID Connector container that runs on a virtual machine. Fig. 2.12 shows a C4 model for the relationship between the developer's local system writing changes to the UCS system used for development.

The following sections describe the setup for development for the ID Connector:

- You have a **`git checkout`** of the `ucsschool-id-connector` repository on your *development machine*.
- To synchronize the changes, you use the script **`devsync`** to synchronize changes on your *development machine*.
- You synchronize the changes to the corresponding `installation` folder of the **`ID Connector`** Docker container.

Hint: If you don't have **`devsync`**, a Univention internal script from the `toolshed` repository, you might as well use **`scp`**, **`rsync`**, or any other transfer mechanism of your liking to copy changes to a remote system.

2.3.1 Machine

To set up the local development environment, run the following commands. They create the directory `venv` with a Python virtual environment with the app and all its dependencies in it.

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

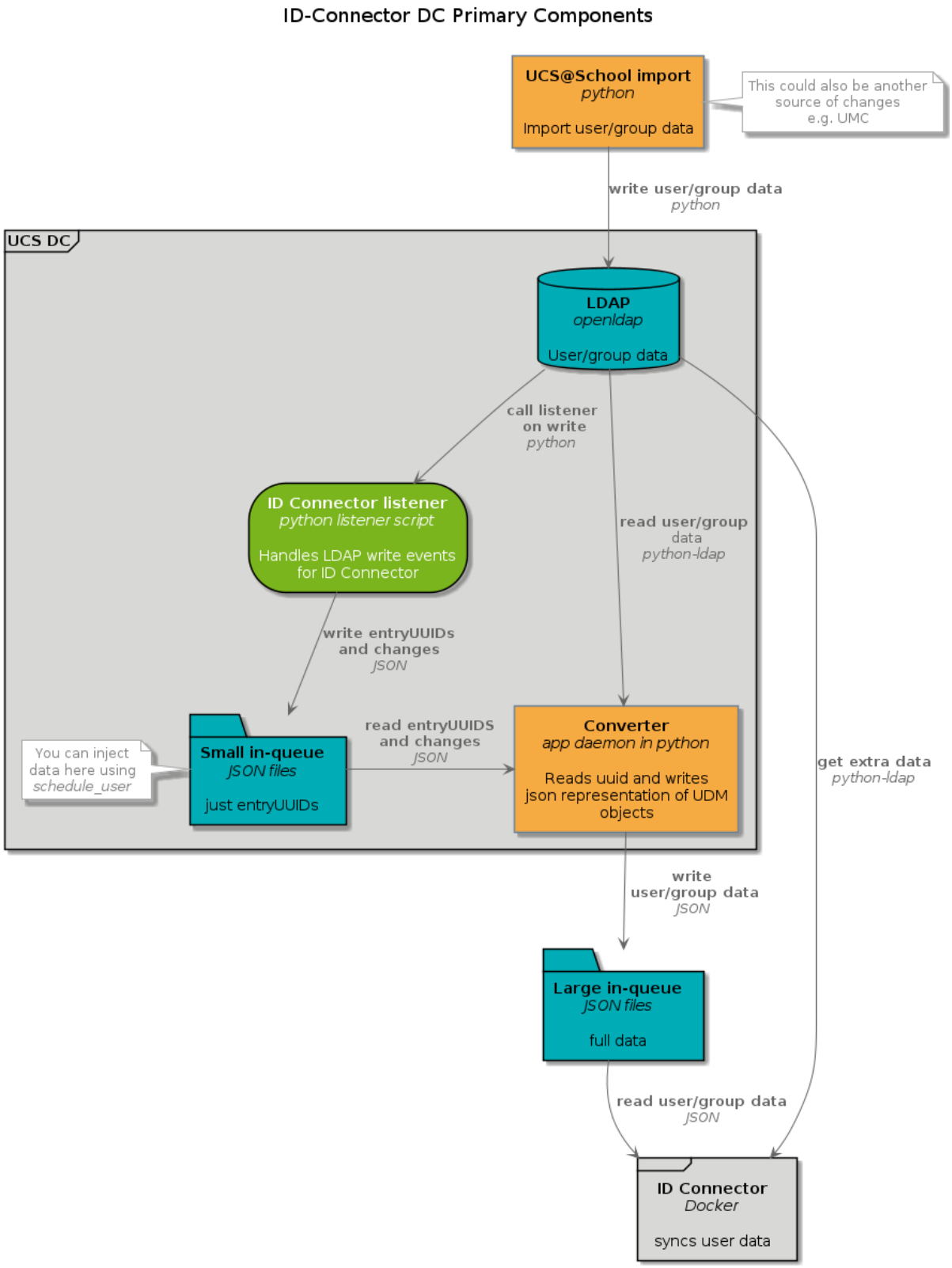

Fig. 2.5: ID Connector Primary Directory Node components

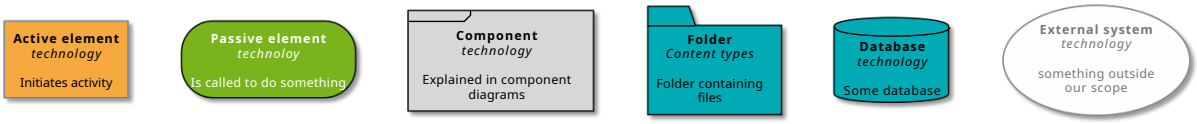

Fig. 2.6: Diagram elements

To *activate* the Python virtual environment in the `venv` directory, run the following command:

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

Warning: All commands in the `Makefile` assume that you **activated** the Python virtual environment.

To see the commands, run **`make`** without argument:

```
$ 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
```

2.3.2 Virtual machine

You need to install the ID Connector app through the Univention App Center on your UCS system for development in a virtual machine.

After installation, the Univention App Center starts the ID Connector container. To enter the container of the app, use the following command:

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

Inside the container, you can use the Python provided by the container's system. Run the following commands from inside the directory `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
```

To synchronize your local working copy into the running ID Connector container on the remote development virtual machine, use the following steps:

1. Stop the ID Connector in it's container on the remote development virtual machine:

   ```
   $ univention-app shell ucsschool-id-connector \
     /etc/init.d/ucsschool-id-connector stop
   ```

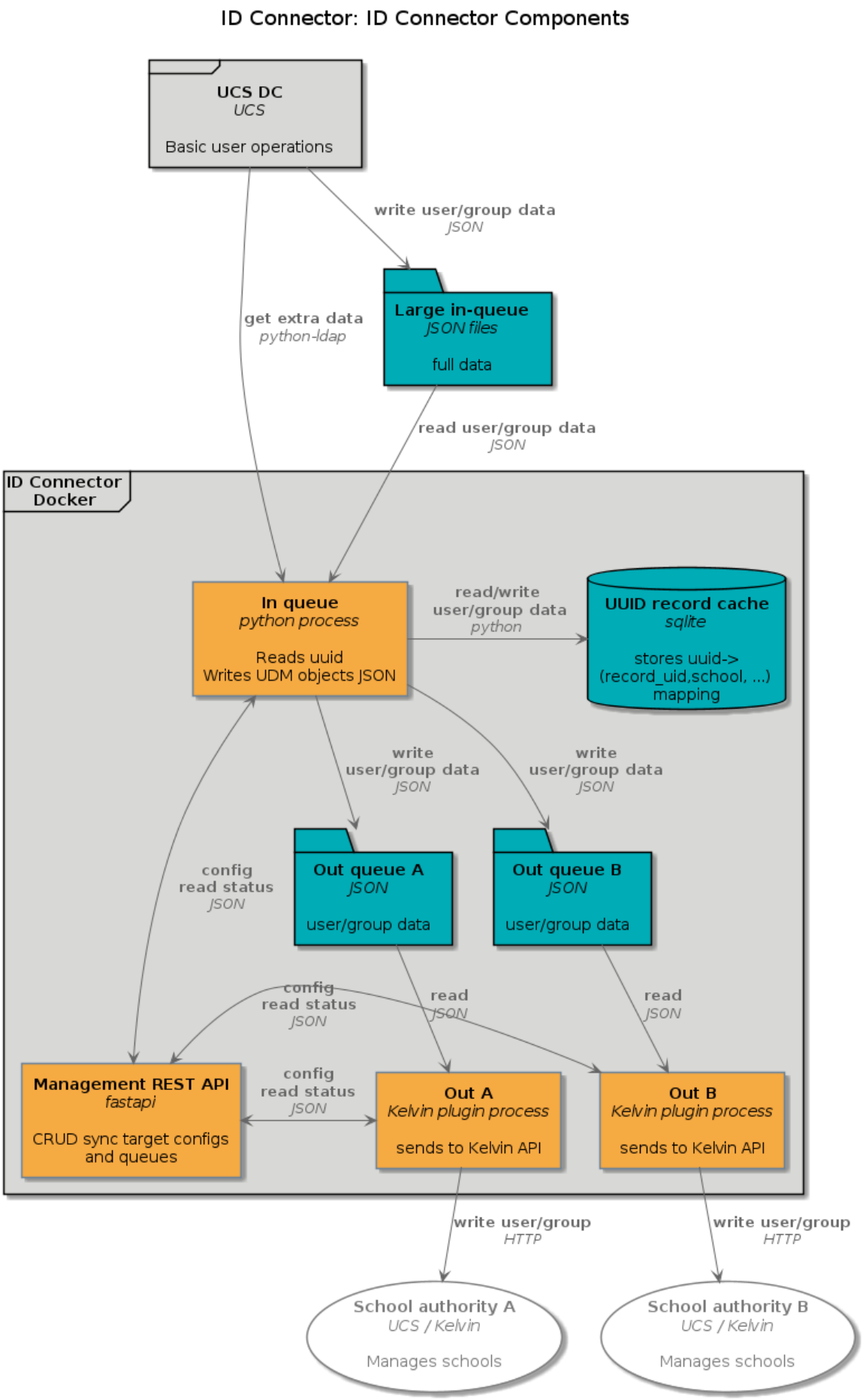

Fig. 2.7: ID Connector components

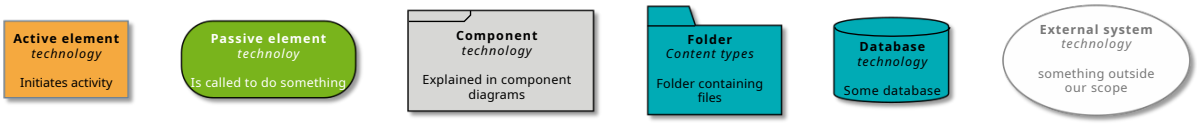

Fig. 2.8: Diagram elements

2. Find out container's ID on the remote development virtual machine:

```
$ docker inspect --format='{{.GraphDriver.Data.MergedDir}}' \
    "$(ucr get appcenter/apps/ucsschool-id-connector/container)"

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

3. On your local developer machine, use container ID to synchronize the local files into the remote container:

```
$ devsync -v src/ \
  "$IP_OF_DEV_VM":/var/lib/docker/overlay2/8dc...387/merged/ucsschool-id-
↪connector/
```

4. Restart and prepare the container for development. The commands perform the following steps:

   On the remote development system:

   1. Enter the ID Connector container.

   In the ID Connector container:

   2. Install the requirements for development.
   3. Install the ID Connector from the source files in development mode.
   4. Start the ID Connector.
   5. Stop the HTTP REST API.
   6. Start the HTTP REST API in *auto-reloading* development mode.
   7. Schedule a user.
   8. Schedule a group.
   9. Schedule a school with a number of parallel tasks.

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

$ python3 -m pip install --no-cache-dir -r src/requirements.txt -r src/
↪requirements-dev.txt

$ python3 -m pip install -e src/

$ /etc/init.d/ucsschool-id-connector restart

$ /etc/init.d/ucsschool-id-connector-rest-api stop

$ /etc/init.d/ucsschool-id-connector-rest-api-dev start

$ src/schedule_user demo_teacher

$ src/schedule_group demo_class

$ src/schedule_school DEMOSCHOOL 32

# DEBUG: Searching LDAP for user with username 'demo_teacher'...
```

(continues on next page)

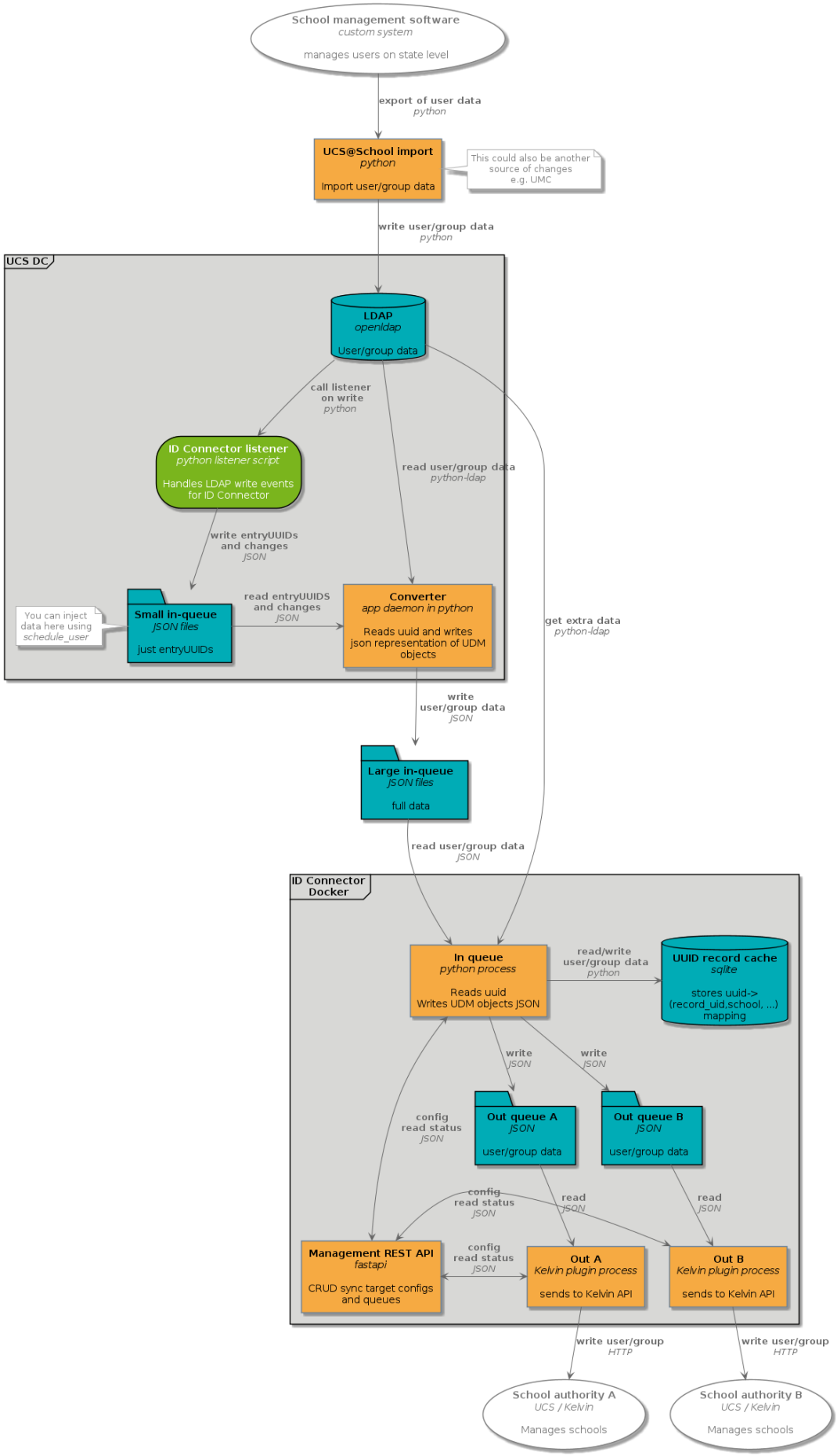

Fig. 2.9: The ID Connector overview in C4 style

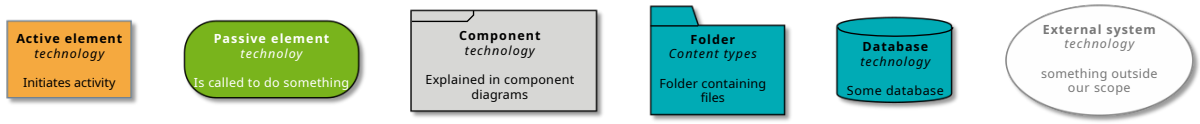

Fig. 2.10: Diagram elements

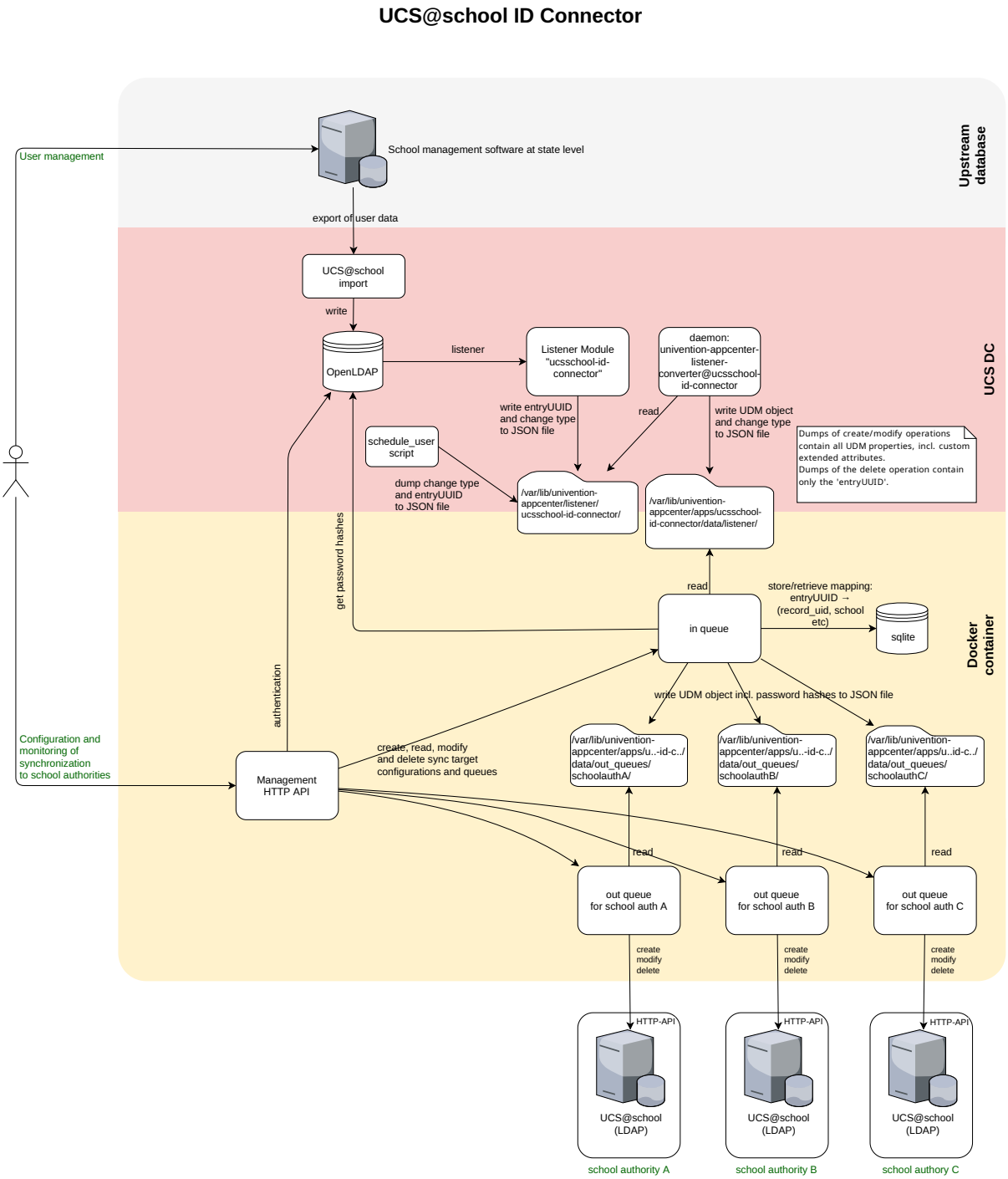

Fig. 2.11: The ID Connector, *not* simplified.

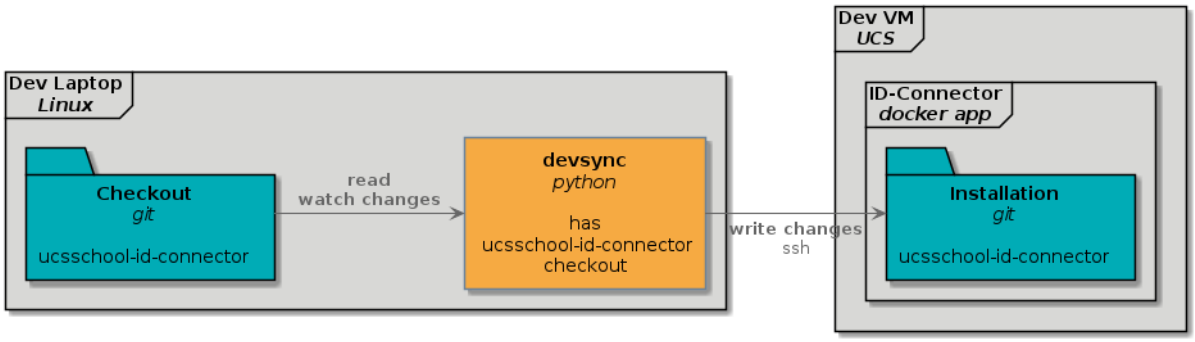

Fig. 2.12: Setup for development

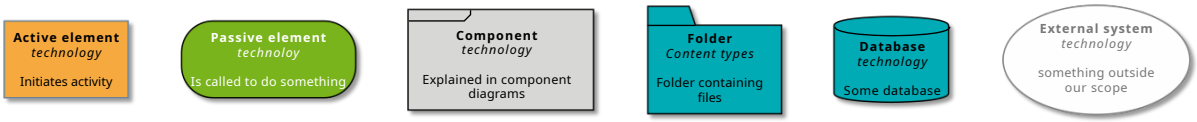

Fig. 2.13: Diagram elements

(continued from previous page)

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

You find logging information in `/var/log/univention/ucsschool-id-connector/queues.log`.

2.3.3 Run unit tests

Unit tests run as part of the *build process* (page 37). To start the units tests manually inside the ID Connector Docker container, run the following commands:

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

2.4 Plugin development

This section describes how to develop a plugin for ID Connector.

2.4.1 How does the plugin system work?

You can enhance the **UCS@school ID Connector** through plugins. ID Connector uses the pluggy[26] plugin system to define, implement, and call plugins.

See also:

To get a short impression on *Pluggy*, have a look at the toy example[27] in the *Pluggy* documentation.

The basic idea for plugins is the following:

- specify hook specifications: callables with the signature you want to have, decorated with a `hook_spec` marker provided by `pluggy.HookspecMarker()`[28].
- write actual hook implementations, also known as *plugins* that ID Connector calls later: callables with the same name and signature as in the specification, but this time decorated with a `hook_impl` marker provided by `pluggy.HookimplMarker()`[29].

The ID Connector system already defines the *`hook_spec`* (page 50) and *`hook_impl`* (page 50) markers. You can use them directly. The same is true for finding and calling your custom plugin.

The key file for ID Connector in this context is `src/ucsschool_id_connector/plugins.py`, where you find the *`hook_spec`* (page 50) and *`hook_impl`* (page 50) markers. In this file you also find the plugin *specifications* as function signatures, decorated with the `@hook_spec` decorator.

The app provides 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.

[26] https://pluggy.readthedocs.io/en/latest/
[27] https://pluggy.readthedocs.io/en/latest/#a-toy-example
[28] https://pluggy.readthedocs.io/en/latest/api_reference.html#pluggy.HookspecMarker
[29] https://pluggy.readthedocs.io/en/latest/api_reference.html#pluggy.HookimplMarker

The ID Connector uses some of the default plugins only if no custom plugins are present. See usages of *filter_plugins()* (page 50) defined in `src/ucsschool_id_connector/plugins.py`:

- *create_request_kwargs()* (page 49)
- *school_authority_ping()* (page 49)
- *handle_listener_object()* (page 49)

2.4.2 A simple custom plugin

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

You find the directory structure for your custom plugins and packages on the host system in `/var/lib/univention-appcenter/apps/ucsschool-id-connector/conf/`. For packages, see the *Advanced example* (page 36) below:

- `/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. For example, you can save the following content 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
```

Validate the queues log file in the directory `/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']
```

The entries tell you that the ID Connector found your plugin `MyPlugin` and the hook implementation for `get_listener_object()`.

2.4.3 Advanced example

In this example, you learn how to additionally:

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

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

```
.../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. The module loader in the `plugin_loader.py` file appends the `packages` directory to the Python `sys.path`[30]. The ID Connector imports packages herein without being *properly* installed.

Listing 2.1: 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
```

Listing 2.2: 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)
```

(continues on next page)

[30] https://docs.python.org/3.8/library/sys.html#sys.path

(continued from previous page)

```
    def dummy_func(self, arg1, arg2):
        """An example hook."""

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())
```

Upon app startup, the ID Connector discovers all plugins and logs them in the log file. You find successful messages like this in the queues log file in the `/var/log/univention/ucsschool-id-connector/queues.log` directory:

```
...
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']
...
```

2.5 Build artifacts

This section describes how to build the ID Connector artifacts, such as the Docker image and the release image.

2.5.1 Build Docker image

The repository contains a `Dockerfile` that you can use to build a Docker image.

Warning: Don't use the image for production. It's suitable for testing and development purposes.

Listing 2.3: Manually start the ID Connector Docker container

```
$ 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 you start the ID Connector Docker container manually as shown in Listing 2.3, and not through the Univention App Center, you need to stop the local firewall with **`service univention-firewall stop`** and

can then access the container through the URL `https://FQDN:8911/ucsschool-id-connector/api/v1/docs`.

You can also:

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

# see the stdout
$ docker logs ucsschool_id_connector

# 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
```

2.5.2 Build release image

Warning: You need to be a software developer at Univention to use this section.

To build a release image, use the following steps:

1. Update the app version in `VERSION.txt`.
2. Add an entry to the changelog in `src/HISTORY.rst`.
3. Adjust the app ini file[31], if needed.
4. The repository pipeline builds the Docker image automatically.
5. Use the dedicated Jenkins job. The job tags the image and also updates the Docker image in the App Provider Portal.
6. Verify that the Jenkins job correctly set the tag for the Docker image of the app in the App Provider Portal.

2.6 Integration tests

Univention has automated integration tests using Jenkins. The Jenkins configuration file locates at https://github.com/univention/univention-corporate-server/blob/5.0-6/test/scenarios/autotest-244-ucsschool-id-sync.cfg. If you want to manually set up integration tests, you need to look there for hints on how to do it.

[31] https://git.knut.univention.de/univention/components/ucsschool-id-connector/-/blob/master/appcenter_scripts/ucsschool-id-connector.ini?ref_type=heads

CHAPTER

THREE

FILE LOCATIONS

This section lists relevant directories and files. **Don't** edit configuration files by hand. *App settings* in the UCS App Center or the *UCS@school ID Connector HTTP API* take care of all configuration.

All important data persists in files on volumes mounted from the UCS host into the Docker container. Therefore, there is no need for distinct backup before an update and a restore afterwards.

3.1 Log files

The directory `/var/log/univention/ucsschool-id-connector` is a volume mounted into the Docker container, so that you can access it from the host.

The directory contains the following files:

- `http.log`: log file of the HTTP-API, both ASGI server and API application.
- `queues.log`: log file of the queue management daemon.
- Previous versions of before mentioned log files with timestamps appended to the filename.

The system's **`logrotate`** settings control log file rotation. For example, to change the rotation cycle to daily for `queues.log`, use the following command:

```
$ ucr set logrotate/ucsschool-id-connector/queues/rotate=daily
```

See also:

See Logging/retrieval of system messages and system status[32] in *UCS Manual* [1].

To view log file output, run the following command:

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

3.2 School authority configuration files

The configuration of the replication targets, such as *school authorities / Schulträger*, locates in one JSON file per configured school authority in the directory `/var/lib/univention-appcenter/apps/ucsschool-id-connector/conf/school_authorities`. Don't create the JSON configuration by hand. Use the *UCS@school ID Connector HTTP API* instead.

[32] https://docs.software-univention.de/manual/5.0/en/computers/basic-system-services.html#computers-logging-retrieval-of-system-messages-and-system-status

3.3 Token signature key

The key for signing the JWTs locates in the file `/var/lib/univention-appcenter/apps/ucsschool-id-connector/conf/tokens.secret`. The app join script creates this file. For more information, see *Installation* (page 6).

3.4 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 file `/var/lib/univention-appcenter/apps/ucsschool-id-connector/conf/ssl_certs/HOSTNAME`. If the *Kelvin client plugin* can't download the certificate automatically, you can manually save it to the preceding location.

3.5 Volumes

The Docker container mounts the following host directories as volumes:

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

CHAPTER

FOUR

EXAMPLE JSON CONFIGURATIONS

This section provides some example configurations.

4.1 Sending system examples

Here you find example configurations for the sending system of an ID Connector setup.

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"
```

(continues on next page)

(continued from previous page)

```
        }
      },
      "sync_password_hashes": true,
      "ssl_context": {
        "check_hostname": false
      }
    }
  }
}
```

4.1.2 School to authority mapping example

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

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",
```

(continues on next page)

(continued from previous page)

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

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 42).

```
{
  "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",
```

(continues on next page)

(continued from previous page)

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

4.2 Receiving system examples

Here you find example configurations for the receiving system of an ID Connector setup.

4.2.1 Mapped UDM properties

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

CHAPTER

FIVE

CHANGELOG

5.1 v2.3.3 (2024-01-11)

- The python package tenacity has been added as additional dependency to properly support the ID-Broker plugin (Issue #101).

5.2 v2.3.2 (2024-01-08)

- The scripts to schedule users, groups and schools have been improved to have a help message (Issue #47).

5.3 v2.3.1 (2023-11-30)

- A new `schedule_group` command has been added. It can be used to force a group to be synced again (Issue #41).
- A new `schedule_school` command has been added. It can be used to force a school to be synced again (Issue #41).
- The ID Connector API patch endpoint for school authorities was fixed (Issue #44).

5.4 v2.3.0 (2023-11-30)

- The rotation of log files is now managed by the UCS host systems `logrotate`. This is also fixing a bug that could lead to missing log entries. (Bug #55983[33]).

5.5 v2.2.8 (2023-08-21)

- ID Connector Kelvin plugin compares OU names case insensitive (Bug #55344[34]).
- Upgrade `Pydantic`, improve `ListenerFileAttributeError` exceptions (Bug #56399[35]).
- The automatic clean up of the ID Connector's `trash` directory now works as intended (Bug #56235[36]). The following issues were fixed:
  - The `listener_trash_cleaner` file is now executable.
  - The `cron daemon` within the Docker-Container runs on startup of the container.

[33] https://forge.univention.org/bugzilla/show_bug.cgi?id=55983
[34] https://forge.univention.org/bugzilla/show_bug.cgi?id=55344
[35] https://forge.univention.org/bugzilla/show_bug.cgi?id=56399
[36] https://forge.univention.org/bugzilla/show_bug.cgi?id=56235

5.6 v2.2.7 (2023-06-22)

- Updated upstream dependencies. A security vulnerability in `starlette` (CVE-2023-30798[37]) was fixed (Bug #56265[38]).

5.7 v2.2.6 (2023-06-14)

- The ID Connector can now be configured to automatically clean up its `trash` directory periodically (Bug #53048[39]). Two new app settings where created:
  - `trash_delete_state` determines if the clean up should be run periodically,
  - `trash_delete_offset` determines after how many days old listener files are be cleaned up.

5.8 v2.2.5 (2023-03-29)

- Boolean attributes are now synced correctly (Bug #54307[40]).

Note: The format of objects which are written by the listener and read by the ID Connector plugins changed from version `2.2.4` and `2.2.5` (cf. Bug #54773[41]). It now has the format of the UDM Rest API objects (e.g. users and groups). Customized plugins might have to be adapted.

5.9 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.

[37] https://cve.mitre.org/cgi-bin/cvename.cgi?name=2023-30798
[38] https://forge.univention.org/bugzilla/show_bug.cgi?id=56265
[39] https://forge.univention.org/bugzilla/show_bug.cgi?id=53048
[40] https://forge.univention.org/bugzilla/show_bug.cgi?id=54307
[41] https://forge.univention.org/bugzilla/show_bug.cgi?id=54773

5.10 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.

5.11 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!

5.12 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`.

5.13 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.

5.14 v2.0.1 (2021-03-04)

- The transfer of Kerberos key hashes has been fixed.

5.15 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.

5.16 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.

5.17 v1.0.0 (2019-11-15)

- Initial release.

CHAPTER

SIX

PLUGIN API

This section describes some of the elements in `src/ucsschool_id_connector/plugins.py` of the ID Connector source code.

6.1 Postprocessing

class ucsschool_id_connector.plugins.**Postprocessing**

Pluggy hook specifications for all hooks modifying data in post processing. The implementations of these hooks need to be registered with a name, since the set of plugins executed can be configured for every school authority individually.

async create_request_kwargs(*http_method: str*[42], *url: str*[43], *school_authority: SchoolAuthorityConfiguration*) → Dict[44][Any[45], Any[46]]

Creates a dictionary the keyword arguments for the http request should be updated with.

The configured `create_request_kwargs` hooks for a given school authority will be executed. The returned dictionaries are used to update the keyword arguments for aiohttp with. Common use cases would be the addition of headers or authentication strategies.

Parameters

- **http_method** – The HTTP method used, e.g. POST
- **url** – The complete URL this request goes to
- **school_authority** – The school authority configuration that this request targets

Returns

The dictionary to update the request keyword arguments with

async handle_listener_object(*school_authority: SchoolAuthorityConfiguration*, *obj: ListenerObject*) → bool[47]

This hook is the entry point for the entire handling logic of `ListenerObjects` in the out queue. All handler hooks that have been registered and appear in a specific school authority configuration are executed. If no registered hook handles the object and thus none returned `True`, an error will be logged.

Parameters

- **school_authority** – The school authority this object is handled for
- **obj** – The ListenerObject to handle

Returns

True if this hook handled the object, otherwise False

async school_authority_ping(*school_authority: SchoolAuthorityConfiguration*) → bool[48]

This hook can be defined to implement a connectivity check to the API of a school authority. If any registered ping hooks for a school authority returns `False`, the communication is considered faulty.

Parameters
school_authority – The school authority to check the connectivity to.

Returns
True if check succeeds, otherwise False

6.2 filter_plugins

`ucsschool_id_connector.plugins.`**`filter_plugins`**(*hook_name: str*[49], *plugins: List*[50]*[str*[51]*]*) → Any[52]

This function returns a HookCaller containing only the implementations of the specified plugins. If the given list is empty, or no specified plugin implements the hook, the default plugin is chosen.

Parameters

- **hook_name** – The hook to be executed
- **plugins** – The plugins to be filtered for

Returns
A `_HookCaller` instance that can be used just like `plugin_manager.hook.hook_name`.

6.3 hook_*

`ucsschool_id_connector.plugins.`**`hook_impl = <pluggy._hooks.HookimplMarker object>`**

Decorator for marking functions as hook implementations.

Instantiate it with a `project_name` to get a decorator. Calling `PluginManager.register()` later will discover all marked functions if the `PluginManager` uses the same project name.

`ucsschool_id_connector.plugins.`**`hook_spec = <pluggy._hooks.HookspecMarker object>`**

Decorator for marking functions as hook specifications.

Instantiate it with a project_name to get a decorator. Calling `PluginManager.add_hookspecs()` later will discover all marked functions if the `PluginManager` uses the same project name.

[42] https://docs.python.org/3.8/library/stdtypes.html#str
[43] https://docs.python.org/3.8/library/stdtypes.html#str
[44] https://docs.python.org/3.8/library/typing.html#typing.Dict
[45] https://docs.python.org/3.8/library/typing.html#typing.Any
[46] https://docs.python.org/3.8/library/typing.html#typing.Any
[47] https://docs.python.org/3.8/library/functions.html#bool
[48] https://docs.python.org/3.8/library/functions.html#bool
[49] https://docs.python.org/3.8/library/stdtypes.html#str
[50] https://docs.python.org/3.8/library/typing.html#typing.List
[51] https://docs.python.org/3.8/library/stdtypes.html#str
[52] https://docs.python.org/3.8/library/typing.html#typing.Any

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 Corporate Server 5.0 Architecture*. Univention GmbH, 2023. URL: https://docs.software-univention.de/architecture/5.0/en/.

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

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

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

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

INDEX