CUE Zipline

# User Guide

1.17.1-2

# Table of Contents


1 Introduction.....5
1.1 Content Store to CUE Print.....5
1.2 Content Store to DC-X.....6
1.3 CUE Print to Content Store.....7
1.4 NewsML Import/Export.....7
1.5 Print Previews in CUE.....7
1.6 Classic/Storyline Conversions.....8
1.7 Clustering.....8
2 Installation.....10
2.1 Ubuntu and Debian.....10
2.2 RedHat.....10
2.3 Configuring CUE Zipline.....11
2.4 Proxying CUE Zipline.....11
2.5 Using Self-Signed Certificates.....12
3 Configuration.....14
3.1 version.....14
3.2 endpoints.....14
3.3 event_listener.....16
3.4 server.....16
3.4.1 converters.....17
3.5 resolver.....18
3.6 logging.....18
3.7 heartbeat.....18
3.8 conversion-templates.....19
3.9 filter.....19
3.10 processors.....20
3.10.1 cue-print Processor.....21
3.10.2 dcx Processor.....24
3.10.3 newsml Processor.....27
3.10.4 newsml-import Processor.....28
3.10.5 External Processors.....29
3.11 copyback.....34
3.12 dcx-converters.....36
3.12.1 Content Duplication.....37

3.13 audit ... 38
3.14 cluster ... 38
3.15 monitoring ... 39
3.16 CUE Print Asset Mapping ... 40
3.17 Mapping Attributes ... 41
3.17.1 Meta Attributes ... 43
3.17.2 Templates ... 45
3.18 locations ... 46
3.19 Environment Variables ... 46
4 Conversion Templates ... 50
4.1 A Templating Example ... 51
4.1.1 The Storyline JSON Structure ... 51
4.1.2 The Target CUE Print Structure ... 54
4.1.3 The Pseudo-Storyline Structure ... 54
4.1.4 The Conversion Templates ... 59
4.2 Handling Other Conversions ... 60
4.3 How To ... 60
4.3.1 Supporting Annotated Image Captions ... 60
5 The Events Plugin ... 65
5.1 Installation ... 65
5.1.1 Ubuntu and Debian ... 65
5.1.2 RedHat and CentOS ... 65
5.2 Configuration ... 66
5.2.1 Amazon SQS ... 66
5.2.2 RabbitMQ (AMQP Protocol) ... 67
6 The Zip Export Plugin ... 69
6.1 Installation ... 69
6.2 Configuration ... 69
6.2.1 Export to Disk ... 70
6.2.2 Export to S3 ... 70
7 The Sophi Plugin ... 72
7.1 Installation ... 72
7.2 Configuration ... 72
7.2.1 Sophi Content Feed Configuration ... 73
7.2.2 Sophi List Updater Configuration ... 75
8 Clustering ... 79
8.1 Front End ... 79

8.1.1 Load Balancing........80
8.2 Core........81
8.2.1 Negotiations........81
8.2.2 Change Log Monitoring........81
8.2.3 Example........82
9 Logging........84
9.1 Configuration........86
10 Monitoring........89
10.1 Requesting a Report........89
10.2 Report Structure........90
10.3 Components........90
10.3.1 Restarts........91
10.3.2 State........91
10.3.3 Processors........92
11 Recording Catch Files........94
11.1 zl-catch........94
11.2 The Catch File API........95

# 1 Introduction

CUE Zipline is a format conversion tool that is primarily intended to provide real-time synchronization of content between CUE Content Store and other Stibo DX systems. Currently, CUE Zipline can provide:

- Automatic synchronization of storyline content (**not** classic rich text stories) from CUE Content Store to CUE Print
- Automatic synchronization of binary assets and /or stories from CUE Content Store to DC-X
- On-demand synchronization of storyline content from CUE Print back to CUE Content Store

CUE Zipline is implemented as a web service that is capable of retrieving/receiving content from CUE Content Store, CUE Print, converting it to the required output format and sending it to the required destination service (CUE Content Store, CUE Print or DC-X).

CUE Zipline monitors activity in CUE Content Store by listening for server-sent events (SSE). When it receives an SSE notification of a significant content change, it:

- Sends a request for the new or modified content to the Content Store
- Performs any data transformation that is required
- `POST`s the transformed content to the required target service (CUE Print or DC-X)

Changes made in CUE Print are not **automatically** copied back to the Content Store. It is only possible to copy back changes made in print packages containing text that originated in the Content Store, and such changes are only copied back if the user explicitly requests it by selecting the **Copy to CUE** option. When this happens CUE Print sends the changed content directly to CUE Zipline in a `POST` request. CUE Zipline then:

- Performs the required format conversion
- `POST`s the converted content to the Content Store

CUE Zipline can, however, also be used for other conversion tasks:

- Automated export of storyline content to NewsML-G2 files
- Automated import of storyline content from NewsML files
- Converting print storylines into CUE Print texts for the purpose of generating print previews.
- Converting rich text-based "classic" content items into storyline content items.

All SSE events in a CUE system are routed through an SSE Proxy, so in order to monitor CUE Content Store, CUE Zipline is connected as a client to the SSE Proxy.

The most important conversions performed by CUE Zipline are described in more detail in the following sections. All the text format conversions are carried out using Jinja2 templates.

## 1.1 Content Store to CUE Print

Copyright © 2020-2023 Stibo DX A/S

All content changes that occur in the CUE Content Store result in the generation of SSE events, which are passed to the SSE Proxy. The SSE Proxy passes on these events to all of its subscribers, one of which is CUE Zipline. CUE Zipline filters these incoming events, ignoring all irrelevant events. If an event describes a change to a print storyline that CUE Zipline is configured to monitor, then CUE Zipline:

- Sends a request for the new or modified content item to the Content Store
- Converts the content item to the format required by CUE Print
- **`POST`**s the converted content item to the CUE Print service

Mapping content to CUE Print may require conversion of multiple content items from CUE Content Store (story, images, graphics, ...) to multiple objects in CUE Print (text object, image/graphics objects). Each object mapping consists of multiple parts: A set of attributes of the target object (identifying meta-data like an object name, online URL, etc.), a text part, representing the article text or an image caption, and possibly a binary part.

While the binary part is transferred without manipulation, the text part is converted in CUE Zipline, using templates, and the attributes are mapped based on configuration.

Default templates that work with standard content types are included with the installation, in the **`/etc/cue/zipline/conversion-templates/cue-print/storyline-to-cue-print`** folder. You may need to modify these templates to work with your content types. For further information see chapter 4.

Likewise, default attribute mappings are included with the installation, in the **`/etc/cue/zipline/asset-mapping`** folder. Modification of these files is less likely to be needed, but it is possible. For further information see section 3.16.

## 1.2 Content Store to DC-X

All content changes that occur in the CUE Content Store result in the generation of SSE events, which are passed to the SSE Proxy. The SSE Proxy passes on these events to all of its subscribers, one of which is CUE Zipline. CUE Zipline filters these incoming events, ignoring all irrelevant events. If an event describes the addition of a classic story, storyline or binary asset (that is, a content item referencing a binary object such as an image, graphic, video, audio file, document, spreadsheet etc.) then CUE Zipline:

- Sends a request for the new content item to the Content Store
- Converts the content item to the format required by DC-X
- **`POST`**s the converted content item to the DC-X service

> The purpose of synchronizing stories and storylines to DC-X is to take advantage of DC-X syndication functionality. Content should not in general be modified in DC-X as any changes made may be overwritten the next time the content item is modified in the Content Store, thereby triggering a synchronization event.

## 1.3 CUE Print to Content Store

In the standard CUE workflow, CUE Content Store is the primary database; the CUE Print server plays a secondary role. In principle, all content editing is done in CUE, including editing of storyline print variants. CUE Print is then mostly used for layout-related adjustments that have no effect on the content. There is therefore no need for SSE-based automated synchronization from CUE Print to the Content Store.

In reality, however, content changes **are** sometimes made in CUE Print (typically last-minute changes) and there is therefore a need to be able to copy changes back to the Content Store (on demand rather than automatically).

For print packages containing texts that originated in the Content Store, CUE Print offers a **Copy to CUE** menu option. Selecting this option causes CUE Print to send an HTTP `POST` request to CUE Zipline containing the current text. CUE Zipline converts the supplied text to the required format and `POST`s the result to the Content Store, thus synchronizing the print variant in the Content Store with the CUE Print package.

Additionally, the ordering workflow in CUE Print can trigger content creation in Content Store if a target container type is selected for the shape(s) content is being ordered for. The `cue-print` product mapping configuration is used to determine which of the chosen containers' first destinations will be used when creating the content.

Default templates that work with standard content types are included with the installation, in the `/etc/cue/zipline/conversion-templates/cue-print/cue-print-to-storyline` folder. You may need to modify these templates to work with your content types. For further information see chapter 4.

## 1.4 NewsML Import/Export

CUE Zipline can be used both for import of content from the NewsML exchange format and for exporting content to NewsML. For import purposes, CUE Zipline can be configured to watch specified folders for the appearance of NewsML files and import any files that appear there. For export, CUE Zipline uses the same SSE-based method as is used for the CUE Print and DC-X conversions, making it possible to automatically export NewsML versions of modified content items. The converted files are not `POST`ed to a remote service, but saved to a specified folder on the local machine.

Default templates that work with standard content types are included with the installation, in the `/etc/cue/zipline/conversion-templates/newsmlg2` folder. You may need to modify these templates to work with your content types. For further information see chapter 4.

## 1.5 Print Previews in CUE

CUE Zipline is used by CUE (the CUE editor) for generating previews of print storylines. When CUE needs to generate a print preview, it `POST`s the content item to CUE Zipline. CUE Zipline then converts the content item into a CUE Print text and returns the text to CUE in its response. CUE then sends the text to CUE Print and receives a preview in response.

Copyright © 2020-2023 Stibo DX A/S 

This feature makes use of the same conversion templates as the section 1.1 conversion.

## 1.6 Classic/Storyline Conversions

CUE Zipline can be used for converting "classic" rich text-based content items to storylines and vice-versa. Assume, for example, that you have a stream of imported content from an external source such as a wire feed, imported as "classic" rich text-based content items, but that you need to be able to open these as storylines in CUE in some circumstances. You can meet such a need by creating an enrichment service that submits the rich text content items to CUE Zipline, which then converts it and returns the resulting storyline. For more information about this use of CUE Zipline, see section 3.4.1.

A very simple set of default templates that works with standard content types is included with the installation, in the **`/etc/cue/zipline/conversion-templates/classic`** folder. You can, however, create your own more sophisticated conversions. For further information see chapter 4.

## 1.7 Clustering

In order to ensure that CUE Print, DC-X and other external systems integrated with the Content Store via NewsML remain synchronized with the Content Store at all times, CUE Zipline needs to be permanently available. You can improve the availability of CUE Zipline by running several instances of it on different hosts in a **cluster**: if one of the instances becomes unavailable (because its host goes offline, for example), then one of the other instances can take over, and synchronization is not interrrupted.

When several instances of CUE Zipline are run as a cluster, one instance is the **active instance**. It monitors the Content Store for changes and exports changed content to CUE Print, DC-X and/or NewsML files, in accordance with its configuration. The other instances are **inactive**. If the active instance becomes unavailable for some reason, then one of the inactive instances will be redesignated as the active instance and continue processing from where the previous active instance stopped. On startup, the instances in a cluster negotiate between themselves to determine which one will be the active instance.

If all the instances in the cluster become unavailable for some reason, then processing will continue from where it was interrupted when the cluster is restarted.

Clustering only affects CUE Zipline's SSE-driven functionality, that is:

- Synchronization of storyline content from CUE Content Store to CUE Print
- Synchronization of binary assets from CUE Content Store to DC-X
- Automated export of storyline content to NewsML

What this means is that inactive instances are not necessarily completely inactive - they will respond as normal to incoming requests from the CUE editor or from CUE Print to perform other functions such as:

- On-demand synchronization of storyline content from CUE Print back to CUE Content Store
- Converting Content Store print storylines into CUE Print texts for the purpose of generating print previews.

Copyright © 2020-2023 Stibo DX A/S

- Converting rich text-based "classic" content items into storyline content items.

Inactive instances can also be used for automated import of NewsML files.

Copyright © 2020-2023 Stibo DX A/S 

# 2 Installation

The CUE Zipline installation procedure is platform dependent – follow the instructions in one of the following sections. If you have a predefined CUE Zipline configuration file, you can streamline the installation process by copying it to **`/etc/cue/zipline/zipline.yaml`** **before** installing. Otherwise, after installing you will need to follow the instructions in section 2.3.

CUE Zipline requires Python version 3.8 or later.

All installation operations must be carried out as **`root`** (it is not always sufficient to use **`sudo`**).

## 2.1 Ubuntu and Debian

To install CUE Zipline:

1. Install the CUE Zipline dependencies:

```
# apt-get install \
  curl \
  gnupg \
  python3-pip \
  python3
```

2. Add the Stibo DX APT source for the appropriate codename to **`/etc/apt/sources.list.d/stibodx.list`**. To add the source for the **`radon`** codename (for example), enter:

```
# echo deb https://apt.escenic.com radon main non-free \
   >> /etc/apt/sources.list.d/stibodx.list
```

3. Add your Stibo DX APT credentials to **`/etc/apt/auth.conf.d/stibodx.conf`**:

```
# vi /etc/apt/auth.conf.d/stibodx.conf
```

```
machine apt.escenic.com
  login username
  password password
```

4. Add the DEB signing key used on the packages in the APT repository, and update your APT cache:

```
# curl --silent http://apt.escenic.com/repo.key | apt-key add -
# apt-get update
```

5. Finally, install CUE Zipline:

```
# apt-get install cue-zipline
```

## 2.2 RedHat

To install CUE Zipline:

1. Install the CUE Zipline dependencies:

```
# yum install -y \
  findutils \
  gcc \
  python38 \
```

```
  python38-devel \
  python38-pip
```

2. On RedHat 7 only (this step is not required on RedHat 8 or later), enter the following command to pull in Python 3.8 from RedHat Software Collections:

```
# yum install -y \
  rh-python38-python \
  rh-python38-python-pip \
  rh-python38-python-devel
```

3. Add the Stibo DX YUM source by entering:

```
# cat > /etc/yum.repos.d/stibodx.repo <<EOF
[stibodx]
name=Stibo DX packages
baseurl=https://user:pass@yum.escenic.com/rpm/
gpgcheck=0
EOF
```

4. Finally, install CUE Zipline:

```
# yum install cue-zipline
```

## 2.3 Configuring CUE Zipline

If you copied a ready-made configuration to **/etc/cue/zipline/zipline.yaml** before installing, then no further steps are required: the CUE Zipline **systemd** service has been automatically started.

If you did not have a ready-made configuration available, a default **zipline.yaml** file will have been created in the **/etc/cue/zipline/** folder, which will need editing. For a detailed description of the configuration file, see chapter 3.

When you have finished editing **zipline.yaml**, you will need to restart CUE Zipline by entering:

```
# systemctl restart cue-zipline
```

## 2.4 Proxying CUE Zipline

Since CUE Zipline exposes both public and private web service end-points, it is strongly advised to install a reverse proxy in front of it, for use by the CUE editor.

The reverse proxy can also function as an SSL/TLS termination point, allowing communication between the CUE editor and CUE Zipline to be secure.

Internal requests, e.g. from CUE Print and trusted enrichment services would still use the direct connection to the server address configured in **zipline.yaml**, which allows access to all web service end-points.

The reverse proxy should pass through requests to **/index.xml**, **escenic/text/***, and **escenic/convert/default** (or **escenic/convert/*** if custom conversions have been configured).

The reverse proxy also needs to set the **X-Forwarded-For**, **X-Forwarded-Proto**, and **X-Real-IP** headers on the request to CUE Print.

Copyright © 2020-2023 Stibo DX A/S

Alternatively, the reverse proxy can set the **Forwarded** header, which combines the information of the other headers.

As an example, if using **nginx** as the reverse proxy, add the following snippet in the **server** configuration:

```
location ~ ^/cue-print-zipline/(index.xml|escenic/text|escenic/convert/default) {
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header Host $http_host;
        proxy_pass http://localhost:12791;
        proxy_pass_header Set-Cookie;
        proxy_read_timeout 185s;
        proxy_set_header Connection '';
        proxy_http_version 1.1;
        chunked_transfer_encoding off;
}
location /cue-print-zipline {
        deny all;
}
```

This example proxies request for the public end-points to port 12791 on the local host (assuming CUE Zipline runs on the same server) and denies access to all other end-points.

## 2.5 Using Self-Signed Certificates

CUE Zipline depends on the curated set of certificate authority (CA) certificates from the Mozilla Project. This means that connecting to servers via HTTPS should work out of the box so long as the server certificates have been acquired from a public certificate authority.

Certificate verification will, however, fail if a server or proxy that CUE Zipline needs to connect to uses a self-signed certificate. To prevent this happening, CUE Zipline must be preconfigured with information about your custom CA certificate. To do this, you need to create a **certificate bundle**, containing all the CA certificates needed by CUE Zipline, both your custom CA certificate and all the public ones. You then need to configure CUE Zipline with the location of the bundle by setting the **REQUESTS_CA_BUNDLE** environment variable.

You can get the path of the file containing CUE Zipline's default CA certificate bundle by entering the following command:

```
$ python -m certifi
```

You must not directly add your custom certificate to this file, because the file is overwritten every time CUE Zipline is upgraded. What you need to do instead is create a new bundle by:

- Copying the file to a new location.
- Appending the content of your custom CA certificate (not the server certificate) to the new file.

For example:

```
$ cat $(python -m certifi) myCA.pem > /path/to/myCABundle.pem
```

Copyright © 2020-2023 Stibo DX A/S

You now have a new certificate bundle containing all the certificates needed by CUE Zipline. Set the **REQUESTS_CA_BUNDLE** environment variable to point to this file, and start CUE Zipline:

```
$ export REQUESTS_CA_BUNDLE=/path/to/myCABundle.pem
$ zipline
```

If **REQUESTS_CA_BUNDLE** is not set when CUE Zipline starts, then it looks for the environment variable **CURL_CA_BUNDLE**. If **CURL_CA_BUNDLE** is also not set, then it uses the Mozilla curated CA certificate set included in the CUE Zipline distribution.

Copyright © 2020-2023 Stibo DX A/S 

# 3 Configuration

CUE Zipline is configured by editing a YAML configuration file, **`zipline.yaml`** located in the **`/etc/cue/zipline`** folder.

YAML is a plain text file format that uses indentation as a means of structuring data. Array values can be specified by preceding each array element with a hyphen followed by a space (you will see examples of this in the following descriptions). The most important thing to remember when editing a YAML file is to never use tabs for indentation, only spaces.

The file has the following top-level entries:

```
version:
endpoints:
event_listener:
server:
resolver:
logging:
heartbeat:
conversion-templates:
filter: []
processors: []
copyback:
dcx-converters:
audit:
cluster:
monitoring:
locations:
```

## 3.1 version

**`version`** specifies the current configuration file version. It must be set to **`3`**:

```
version: 3
```

## 3.2 endpoints

**`endpoints`** contains the URLs and credentials CUE Zipline needs to access Content Store, CUE Print and DC-X:

```
endpoints:
    content_store:
        url: content-store-host/webservice/
        user: content-store-user
        password: content-store-password
        system-id: system-id
    cue_print:
        url: cue-print-host/newsgate-cf/
        user: cue-print-user
        password: cue-print-password
    dcx:
```

Copyright © 2020-2023 Stibo DX A/S

```
        url: dcx-endpoint-url
        user: dcx-user
        password: dcx-password
    dpres_cook:
        url: cook-endpoint-url
        user: cook-user
        password: cook-password
```

A **dcx** entry is only required if your installation actually includes a DC-X system and you intend to use CUE Zipline for synchronization of either binary assets or stories/storylines.

A **dpres_cook** entry is only required if your installation configures an attribute mapping referencing the "cook-url" meta attribute.

Each endpoint supports the following properties:

**url (required)**
The URL of the service end-point.

**user (required)**
The user name to use for authentication against the service endpoint.

**password (optional, one of password or password_file must be specified)**
The password to use for authentication against the service end-point.

**password_file (optional, password or password_file must be specified)**
The path of a local file that contains the password to use for authentication against the service end-point. If given, the contents of the file is read using the systems character encoding and used for password.

The **content_store** endpoint also supports the following additional property:

**system-id (optional)**
A global identifier for this Content Store endpoint. If specified, the identifier is supplied when uploading assets to both CUE Print and DC-X endpoints. It is useful at installations where CUE Print and/or DC-X systems are connected to more than one Content Store, ensuring that both assets and asset usage are correctly attributed. The **system-id** set here can be overridden in the **cue-print** and **dc-x** processor definitions (see section 3.10.1 and section 3.10.2), allowing different IDs to be used for each system if required.

For backwards compatibility, the property **passwd** is also supported in an endpoint configuration. It has the same meaning as **password**.

CUE Zipline supports the conventional **http_proxy**, **https_proxy**, and **no_proxy** environment variables, if endpoints need to be connected to via a proxy.

The **http_proxy** and **https_proxy** variables define the proxies to use for HTTP and HTTPS connections. For example:

```
$ export HTTP_PROXY=http://insecure.proxy.lan

$ export HTTPS_PROXY=https://secure.proxy.lan
```

It is possible to use a different protocol when connecting to the proxy from what is required for the backend. For instance, using HTTP to talk to the proxy, while using HTTPS to connect to the backend:

```
$ export https_proxy=http://proxy.lan
```

If a proxy password is required for the connection, then it should be part of the URL configured for the proxy. For example:

```
$ export https_proxy=https://zipline:password@proxy
```

The **no_proxy** variable defines a comma-separated list of host names that can be accessed without passing through a proxy. For instance:

```
$ export no_proxy=localhost,127.0.0.1,*.internal.lan
```

All parameters are supported in both all lowercase and all uppercase variants. So configuring either **https_proxy** or **HTTPS_PROXY** will cause CUE Zipline to use the proxy. If both versions are set, they should be set to the same value.

## 3.3 event_listener

**event_listener** is an **optional** entry that can be used to specify a URL and credentials for accessing the SSE Proxy. It is not required since CUE Zipline is capable of discovering the SSE Proxy itself, and if no other login credentials are specified, it will use the Content Store login credentials specified under the **endpoints** entry. You can, however, user the **event_listener** entry to override the discovery process, or specify alternative credentials:

```
event_listener:
    sse_endpoint:
        url: sse-proxy-endpoint-url
        user: content-store-user
        passwd: content-store-password

    progress_path: progress-file-path
```

The individual parameters should be set as follows:

**sse_endpoint (optional)**
: If the default lookup of the server sent events (SSE) endpoint isn't sufficient, this is where a custom URL (and possibly user name and password) can be configured.

  If a URL is configured here, CUE Zipline will not perform an automatic lookup through CUE Content Store

**progress_path (optional)**
: The path of a file in which CUE Zipline stores its current read positions in CUE Content Store change logs between restarts.

  If no path is configured here, CUE Zipline uses the value of the **ZL_EVENT_LISTENER_PROGRESS_PATH** environment variable or, if that is undefined or empty, the default path **/var/cache/zipline/task-progress.json**.

## 3.4 server

**server** contains settings for CUE Zipline's built-in web server. The server should be configured to only accept requests from appropriate sources.

```
server:
```

Copyright © 2020-2023 Stibo DX A/S

```
    address: 127.0.0.1
    port: 12791
    context-path: /cue-print-zipline
    accepted-origins:
        - https?://localhost(:[0-9]+)?
        - https?://([^.]+\.)*my-cue-domain.com(:[0-9]+)?
    converters:
```

The individual parameters should be set as follows:

**`address` (optional, default: `127.0.0.1`)**
: The IP address of the network interface on which the server is to listen. For production purposes It should usually be set either to the address of a network interface or **`0.0.0.0`**.

**`port` (optional, default: `12791`)**
: The port on which the server is to listen.

**`context-path` (required)**
: The name of the CUE Zipline service (the first part of the service URL after the host name and port). By convention it should be set to **`/cue-print-zipline`**.

**`accepted-origins` (required)**
: An array of regular expressions defining the sources from which the server will accept requests. The list should usually be limited to the local host and the CUE (editor)'s domain (needed to support CUE Zipline's support for print previews in CUE).

**`converters`**
: See section 3.4.1.

### 3.4.1 converters

CUE Zipline provides a default service for converting simple rich text-based content items to storylines. This default converter can be used by any client with access to CUE Zipline. An enrichment service, for example, can convert a rich text-based content item to a storyline by **`POST`**ing the content item to **`https://`***zipline-host***`/cue-print-zipline/escenic/convert/default`**. A rich text content item POSTed to this URL will be passed to the Jinja2 template **`/etc/cue/zipline/conversion-templates/classic/classic-to-storyline/storyline.json`**, which returns a JSON structure containing a **pseudo-storyline**, that can be used to construct a storyline content item (see below for more about pseudo-storylines).

In some cases, however, this simple conversion might be insufficient. The imported content items might come from different sources, and therefore be imported as different content types with different fields. In this case you would then want to define your own templates for converting the different content types. Alternatively you might want to set up a converter to convert content items in the opposite direction – storyline to classic.

The **`server/converters`** section of **`zipline.conf`** allows you to expose such specialized converters on their own URLs. For example:

```
server:
    converters:
        ap:
            template_dir: /etc/cue/zipline/myproject/classic_converters
            template: ap_converter.json
        ntb:
            template_dir: /etc/cue/zipline/myproject/classic_converters
            template: ntb_converter.json
```

For each custom converter you add an entry under **server/converters**. The entry name you use becomes the final segment of the converter URL. The entry name **ap** in the example above will be exposed as **https://***zipline-host***/cue-print-zipline/escenic/convert/ap**. Each such entry must have two child settings:

**template_dir (required)**
: The absolute path of the folder containing the custom template.

**template (required)**
: The name of the custom template.

Note that these transformations convert between classic content items in the form of Atom entries as returned by the Content Store web service and **pseudo-storylines**. A pseudo-storyline:

- Only contains what appears in the storyline editor in CUE – it does not include any of the metadata and other fields that make up a complete content item.
- Contains a modified version of the storyline data structure. It is easier to convert between the pseudo-storyline structure and XML formats such as NewsML, CUE Print text XML and classic CUE content items than it is to convert directly between the true storyline format and such XML formats.

These converters therefore only provide a partial conversion between classic content items and storyline content items. CUE Zipline does, however provide an API for converting between the storyline and pseudo-storyline formats.

## 3.5 resolver

CUE Zipline's resolver is responsible for retrieving the content items referenced in incoming events, and all the information needed to deal with them. The **resolver** section of the configuration file is optional, but can be used to limit the size of the resolver's cache:

```
resolver:
  cache:
    max_size:
```

**max_size** specifies the maximum number of elements the resolver cache may hold. The default is **1000**.

## 3.6 logging

The **logging** section is used to configure the log messages output by CUE Zipline. It contains a standard Python logging configuration as described here. For further information about CUE Zipline logging, see chapter 9.

## 3.7 heartbeat

CUE Zipline sends a heartbeat request at regular intervals to all the services it is connected to (the Content Store, CUE Print and DC-X). If it does not get a response then the service is marked as

Copyright © 2020-2023 Stibo DX A/S

currently unavailable. The **heartbeat** section of the configuration file contains the following two properties:

**period (optional, default: 500)**
: The interval between heartbeats, specified in seconds.

**timeout (optional, default: 5)**
: The length of time CUE Zipline waits for a response before marking a service as unavailable, specified in seconds.

## 3.8 conversion-templates

**conversion-templates** contains a single property, **path**, that specifies the location of all the Jinja2 templates used to carry out the various format conversions performed by CUE Zipline. The location is specifed as an absolute or relative path (relative to the location of the configuration file):

```
conversion-templates:
  path: ./conversion-templates
```

For further information about the templates stored in this folder, see chapter 4.

## 3.9 filter

**filter** is used to filter the stream of incoming events from the SSE Proxy. It contains an array of filters that are used to select the events that CUE Zipline will submit for processing: all other events are ignored.

```
filter:
  - publication
    - tomorrow-online
    - tomorrow-today
    - living-tomorrow
  - type:
    - storyline
    - print
    - print-story
  - story_type:
    - storyline
  - state
    - published
    - approved
  - created-within:
      weeks: 4
```

The filter array may contain any of the following filter types:

**publication**
: Contains an array of publication names. Only events affecting one of the listed publications are selected.

**type**
: Contains an array of content type names. Only events affecting one of the listed content types are selected.

**`story_type`**
Contains an array of **story types** (**`storyline`** or **`classic`**). Only events affecting one of the listed story types are selected.

**`state`**
Contains an array of workflow state names. Only events affecting content items in one of the listed states are selected.

The workflow state names can be specified as a YAML/JSON array of names or as a string, containing a comma-separated list of state names.

**`created_within`**
Only events affecting content items created within the specified period are selected. You can specify the required period in **`weeks`**, **`days`**, **`hours`**, **`minutes`** or **`seconds`**.

Only events which satisfy **all** of the filter conditions listed in the **`filter`** array are selected. You can, however, control exactly how the filter conditions are combined by making use of two additional filters:

**`all`**
Contains an array of sub-filters. Only events selected by all the sub-filters are selected. In the following example, an event will only be selected if it affects a content item of the type **`storyline`** that is in the state **`published`**.

```
- all:
  - type:
    - storyline
  - state:
    - published
```

**`any`**
Contains an array of sub-filters. Any event selected by at least one of the sub-filters is selected. In the following example an event will be selected if it affects a content item that is either of the type **`archive`** or was created within the last 24 hours.

```
- any:
  - created-within:
    hours: 24
  - type:
    - archive
```

## 3.10 processors

**`processors`** contains an array of processor definitions. All the events selected in the **`filters`** section of the configuration file are passed to all the processors defined in this section (except for the **`newsml-import`** processor). When a processor receives an event, it:

- Performs additional filtering to determine what do with the event (process it in some way or ignore it)
- If the event is to be processed:
  - Carries out any necessary transformations on the content referenced by the event
  - Sends the transformed content to the appropriate destination

The following types of processor may be defined:

Copyright © 2020-2023 Stibo DX A/S

**`type: cue-print`**
This processor type converts the content item referenced in an event to a CUE Print text item or asset and submits it in a **`POST`** request to the CUE Print server.

**`type: dcx`**
This processor type converts the content item referenced in an event to a DC-X asset and submits it (along with its metadata) in a **`POST`** request to the DC-X server.

**`type: newsml`**
This processor type converts the content item referenced in an event to a NewsML document and writes it to file.

**`type: newsml-import`**
This processor is an import processor rather than an export processor, and therefore does not receive or respond to events. Instead, it monitors a specified folder for changes, and when NewsML files appear there, imports them as storyline content items.

It is also possible to define **external processors**. Unlike the internal processors provided by CUE Zipline, external processors do not have a **`type`** property.

All these processor types (including external processors) are described in detail in the following sections.

### 3.10.1 cue-print Processor

A **`cue-print`** processor converts the content items referenced by the events it receives to CUE Print texts or assets and uploads them to CUE Print. A **`cue-print`** processor definition consists of the following entries:

```
- type: cue-print
  system-id:
  asset-mapping:
  conversion-templates:
  product-mapping:
  desk-mapping:
```

**`type`** must be set to **`cue-print`**. The other entries are:

**`system-id`** **(optional)**
May be used to identify the source system when attaching multiple CUE systems to a single CUE Print system. The expected value is a string, used by CUE Print to identify the CUE system to send content back to.

If this property is not specified, then the ID is read from the global **`ZL_CUE_PRINT_EXTERNAL_SYSTEM_ID`** environment variable. If this environment variable is not set, then the global **`system-id`** specified as part of the Content Store endpoint definition (see section 3.2) is used as a fallback. If no **`system-id`** is specified there either, then no external system ID is reported to CUE Print.

**`asset-mapping`**
May optionally be used to override the location of the asset mapping configuration files for CUE Print. The value is a path to a folder containing the asset mapping configuration files. The path is resolved relative to the directory containing the main CUE Zipline configuration file.

Copyright © 2020-2023 Stibo DX A/S

**`conversion-templates`**

This configuration option can be used to define a custom location for conversion templates for this particular **`cue-print`** processor, if it requires templates that are different from the default templates used by CUE Zipline.

The configuration options for this property are the same as for the corresponding property on the global configuration level and are described in section 3.8.

**`product-mapping` (required)**

See section 3.10.1.1.

**`desk-mapping` (required)**

See section 3.10.1.2.

#### 3.10.1.1 product-mapping

The **`product-mapping`** section of a **`cue-print`** processor performs two functions:

- It selects which incoming events will be handled, based on publication and content type
- It defines how to upload the content items referenced by these events, based on the same criteria

```
product-mapping:
  - publication:
      - tomorrow-today
      - living-tomorrow
    content:
      - print
      - print-story
    assets:
      image:
        - picture
        - graphic
  - publication:
      - tomorrow-online
    content: []
    assets:
      image:
        - picture
        - graphic
```

The **`product-mapping`** section contains an array, each element of which contains the following entries:

**`publication` (required)**

An array of publication names. Only content from these publications will be processed.

**`content` (required)**

An array of content type names. Only content of these types will processed. You should only specify "text" content types here (i.e. stories not images, graphics, videos etc.)

**`assets` (required)**

Contains the asset type name **`image`**, which in turn contains an array of content type names. You should only specify image/graphic content types here. In future versions of CUE Zipline, other asset types such as videos, documents and spreadsheets may be supported.

If an event matches both a **`publication`** and a **`content`** entry in the same group, then the content item it references will be converted to a CUE Print text and **`POST`**ed to the CUE Print server. If an event matches both a **`publication`** and an **`assets`** entry in the same group, then the content item it

Copyright © 2020-2023 Stibo DX A/S

references will be converted to a CUE Print asset of the appropriate type and **POST**ed to the CUE Print server. In the example shown above for example, **print** and **print-story** content items that belong to either **tomorrow-today** or **living-tomorrow** will be **POST**ed to the CUE Print server as texts. **picture** and **graphic** content items that belong to the same publications, however, will be **POST**ed to the CUE Print server as image assets.

If the product mappings are the same for all publications, then the array may have only one entry (as in the example shown above). If, however different groups of publications require different mappings, then multiple entries will be needed.

#### 3.10.1.2 desk-mapping

The **desk-mapping** section of a **cue-print** processor determines which CUE Print newsroom, product and desk/subdesk a content item is sent to, based on its Content Store home section.

```
desk-mapping:
  - newsrooms:
      tomorrow-today: Tomorrow
      living-tomorrow: Living
    products:
      tomorrow-today: TT
      living-tomorrow: Living
    desks:
      ece_frontpage:
        desk: Home
        layout: Frontpage
      news:
        desk: News
        layout: News
      politics:
        desk: News
        subdesk: Politics
        layout: News
```

The **desk-mapping** section contains an array, each element of which contains the following entries:

**newsrooms (required)**
One or more mappings from Content Store publication names (specified as YAML keys) to CUE Print newsroom names (specified as values). In the example shown above, all content items belonging to the Content Store **tomorrow-today** publication will be directed to the CUE Print **Tomorrow** newsroom, and all content items belonging to the Content Store **living-tomorrow** publication will be directed to the CUE Print **Living** newsroom.

**content (required)**
One or more mappings from Content Store publication names (specified as YAML keys) to CUE Print product names (specified as values). In the example shown above, all content items belonging to the Content Store **tomorrow-today** publication will be directed to the CUE Print **TT** product, and all content items belonging to the Content Store **living-tomorrow** publication will be directed to the CUE Print **Living** product.

**desks (required)**
One or more mappings from Content Store section unique names to CUE Print **desk** names, **layout** names and optionally **subdesk** names. The mappings take advantage of standard Content Store section inheritance rules. The first entry in the above example defines a default mapping for all the sections in a Content Store publication. Content items that belong to the root section (**ece_frontpage**) **and all its subsections** will be assigned to the CUE Print

**`Home`** desk and given a **`Frontpage`** layout. This mapping can, however, be overridden for specific subsections. In the above example such overrides have been created for the **`news`** and **`politics`** sections.

These override mappings are in turn inheritable and can also be overridden. Content items that belong to the **`news`** section and all its subsections will be assigned to the CUE Print **`News`** desk and given a **`News`** layout. Content items that belong to the **`politics`** section and all its subsections will be assigned to the **`Politics`** subdesk of the **`News`** desk and given a **`News`** layout in CUE Print.

If the desk mappings are the same for all publications, then the array may have only one entry (as in the example shown above). If, however different groups of publications require different mappings, then multiple entries will be needed.

### 3.10.2 dcx Processor

A **`dcx`** processor uploads the content items referenced by the events it receives to DC-X. A **`dcx`** processor has the following properties:

```
- type: dcx
  cache:
    max_size:
  cue_web:
    info:
      view:
        label: View
        link_text: Browse
      edit:
        label: Edit
        link_text: Open in CUE
  system-id:
  upload:
```

> If you are using CUE Zipline to upload both binary assets and stories/storylines to DC-X, then you need to define two separate processors, one to handle the binary assets and one to handle the stories/storylines, since the configuration requirements are different in each case.

**`type`** must be set to **`dcx`**. The other entries are:

**`cache`** **(optional)**
May optionally be used to specify cache settings. Currently the only setting available is:

**`max-size`** **(optional, default: `10000`)**
The maximum number of elements the upload cache may hold.

**`cue_web`** **(required)**
Must contain the CUE editor's endpoint URL.

**`system-id`** **(optional)**
May be used to identify the source system when attaching multiple CUE systems to a single DC-X system. The expected value is a string, used by DC-X to identify the source CUE system. The specified string is appended to DC-X assets' ContentStoreId field values (in the form *asset-id*@*system-id*) and used to set the value of DC-X usage records' **`PROD_SYS_ID`** properties.

If this property is not specified, then the ID is read from the global **`ZL_DCX_EXTERNAL_SYSTEM_ID`** environment variable. If this environment variable is not set, then the global **`system-id`** specified as part of the Content Store endpoint definition (see

section 3.2) is used as a fallback. If no **system-id** is specified there either, then the default system ID **CUE** is reported to DC-X.

**upload (required)**
See section 3.10.2.1.

#### 3.10.2.1 upload

The **upload** property of a **dcx** processor serves two purposes:

- It selects which incoming events will be handled, based on publication, content type and state
- It specifies how the selected events are to be handled

```
- filter:
    publications:
      - tomorrow-online
    content-types:
      - picture
      - graphic
    states:
      - approved
      - published
  content:
  folder: native
```

The **upload** property is an array, each element of which contains the following entries:

**filter (optional, default: no additional filtering)**
A DC-X-specific filter that works in exactly the same way as the global filter described in section 3.9. It performs additional filtering to select only those events that are to be handled by the DC-X processor.

> Note that if **deleted** is included the list of states, then whenever a matching content item is deleted in the Content Store, it will also be deleted from DC-X.

**content (required)**
Contains a required **tags** property that defines the details of how content is uploaded to the DC-X server, in the form of mappings between content item fields and DC-X tags. For details, see section 3.10.2.1.1. If the content types to be uploaded are stories/storylines rather than binary assets, then **content may** also contain an **image-container** property (see section 3.10.2.1.2).

**folder (optional, default: story)**
The name of an existing import folder in the DC-X system to which stories will be uploaded. You are recommended to use the default folder name **story**, which exists in a standard DC-X installation.

**upload-configuration (optional, default: uploadconfig_cue_dam_generic)**
The name of an existing upload configuration in the DC-X system to be used for uploading binary assets. You are recommended to use the default configuration name **uploadconfig_cue_dam_generic**, which exists in a standard DC-X installation.

**document-type (required for story/storyline uploads, not used for binary asset uploads)**
The name of a document type defined in DC-X. Uploaded stories/storylines will be created as documents of this type. This property is not used when uploading binary assets.

#### 3.10.2.1.1 tags

The tag mappings specified in a **`tags`** property consist of:

- A **`name`** property identifying a DC-X tag
- A second property specifying how the DC-X tag is to be set

```
tags:
- name: Creator
  first-of:
  - field: byline
  - meta: author
  - meta: creator
- name: Title
  meta: title
- name: body
  template: >
    {%- raw %}
    <p>{{caption}}</p>
    {%- endraw %}
  context:
  - name: caption
    field: caption
- name: Provider
  first-of:
  - field: credit
  - meta: organizational-unit
```

The following variations are possible:

- ```
  - name: Creator
    field: byline
  ```

  Assign the value of the uploaded content item's **`byline`** field to the DC-X **`Creator`** tag.
- ```
  - name: Creator
    meta: creator
  ```

  Assign the value of the uploaded content item's **`creator`** metadata field to the DC-X **`Creator`** tag.
- ```
  - name: Creator
    first-of:
    - field: byline
    - meta: author
    - meta: creator
  ```

  Read the fields listed under **`first-of`** in the specified order. Use the first one that contains a value to set the DC-X **`Creator`** tag.
- ```
  - name: body
    template: >
      <p>{{caption}}</p>
    context:
      - name: caption
        field: caption
  ```

  Use the result of executing the specified Jinja2 template to set the DC-X **`body`** tag. The **`context`** property can be used to define the variables that will be available to the template. These variables can be assigned values in exactly the same way as values are assigned to DC-X tags. So in this

Copyright © 2020-2023 Stibo DX A/S

example, the **{{caption}}** variable will be replaced with the content of the uploaded content item's **caption** field.

#### 3.10.2.1.2 image-container

The **image-container** property is only used when uploading stories or storylines, and it is optional. It is used to define the details of how images in stories/storylines are handled, in the form of mappings between content item fields and DC-X image container tags. If no **image-container** property is specified then the images are stored as related items of the story in DC-X. The tag mappings are defined in exactly the same way as for uploaded binary assets (see section 3.10.2.1.1).

Here is an example **image-container** definition:

```
image-container:
  - content-type: picture
    tags:
      - name: ImageCaption
        first-of:
          - type: image
            field: caption
          - summary-field: caption
          - field: caption
  - content-type: graphic
    tags:
    first-of:
      - type: image
        field: caption
      - summary-field: caption
      - field: caption
```

### 3.10.3 newsml Processor

A **newsml** processor exports the content items referenced by the events it receives to NewsML files (which may be used as input to external systems). A **newsml** processor definition contains the following properties:

```
  - type: newsml
    filter:
    output:
    - type: file
      output_dir:
      encoding:
      file_name_template:
    download_dir:
```

**type** must be set to **newsml**. The other properties are:

**filter (optional, default: no additional filtering)**
: A NewsML-specific filter that works in exactly the same way as the global filter described in section 3.9. It performs additional filtering to select only those events that are to be handled by the **newsml** processor.

**output (optional, default: one type=file element with default settings)**
: An array, each element of which contains settings for a different output method. Currently, however, only one output method is supported, so the array will never contain more than one element.

Copyright © 2020-2023 Stibo DX A/S

**`type` (required)**
The only allowed value is `file`, indicating that the NewsML output will be written to file.

**`output_dir` (optional, default: `/var/backup/cue/zipline`)**
The absolute path of the folder to which output NewsML will be written.

**`encoding` (optional, default: `utf-8`)**
The encoding to be used in the output NewsML file (specified in its XML declaration).

**`file_name_template` (optional, default: `{{id}}.xml`)**
A Jinja2 template defining how the output NewsML files will be named. The following properties are available for use in the templates:

`id` (content item ID)
`year`
`month`
`day`
`hour`
`minute`
`second`
`micro`

So a template setting such as `{{year}}/{{month}}-{{day}}-{{id}}.xml` would result in file paths like this: `2020/06-30-9387.xml`.

**`download_dir` (optional, default: `/tmp/cue/zipline/newsml`)**
The absolute path of the folder to which downloaded binary files will be written. (If an image content item, for example, is selected and converted to NewsML format, then the image binary file it references is downloaded to this folder.)

### 3.10.4 newsml-import Processor

The NewsML import processor is different from all the other processors in that it imports data into the Content Store rather than exporting it, and is therefore not driven by Content Store events. Instead, the NewsML import processor monitors specified import folders and imports any NewsML-G2 files that appear in them.

```
- type: newsml-import
  target:
    publication: tomorrow-online
    section: ece_incoming
  content-types:
      images: picture
      stories: story
  watch_dirs:
  - path: /var/spool/newsml/import
    files:
    - *.xml
    - *.ml
  download_dir: /tmp/cue/zipline/newsml
```

`type` must be set to `newsml-import`. The other entries are:

**`target` (required)**
Contains two properties specifying the publication name and section unique name to be used to identify the home section of created content:

Copyright © 2020-2023 Stibo DX A/S

`publication` **(required)**
The name of the publication to import into.

`section` **(required)**
The unique name of the section, in the targeted publication, to use as home section of the imported content.

`content-types` **(required)**
Contains two properties specifying the content types to be used for importing content to the target publication:

`images` **(required)**
The name of the content type to be used for importing images.

`stories` **(required)**
The name of the content type to be used for text content. Only storyline content types are supported, not classic rich text-based content types.

`watch_dirs` **(required)**
An array, each element of which specifies a folder in which to look for NewsML files to import. Each element may contain the following properties:

`path` **(required)**
The absolute path of a folder in which to look for NewsML files.

`files` **(optional, default: `*.xml`)**
An array of file name patterns to use when looking for files to import.

`download_dir` **(optional, default: `/tmp/cue/zipline/newsml`)**
The absolute path of a folder to be used by CUE Zipline to hold temporary files downloaded from the Content Store during the import process.

### 3.10.5 External Processors

External processors are designed to allow implementation of custom event handlers for a project. The implementation must be written in Python, but is otherwise unrestricted.

External processors are executed in an external process (outside the core Zipline process) to ensure that any problems in the processor don't affect the core event processing done in CUE Zipline.

Unlike the internal processors provided by CUE Zipline, external processors do not have a **`type`** property. The "external" type is implied.

However, a **`name`** property is required, and must be unique across all the configured external processors.

The properties of an external processor are:

`name` **(required)**
Defines a name for the external processor. The name must be unique across all external processors configured for this CUE Zipline cluster.

`source` **(required)**
The **`source`** property defines the event source configuration for this external processor. CUE Zipline monitors this source for events and passes events on to the external processor.

Copyright © 2020-2023 Stibo DX A/S 

**`filter` (optional)**
You can define a filter in order to reduce the number of events passed to the external processor. The supported options for this filter are defined in section 3.9. If no filter is defined, then all events from the configured content update change logs are passed to the external processor.

**`agent` (required)**
Must provide sufficient information about the external processor's event handler (the agent) for CUE Zipline to be able to instantiate it and pass events to it.

It also includes a **`configuration`** object that is passed to the event handler during initialization.

#### 3.10.5.1 `source`

The **`source`** configuration object contains either an **`events`** property, which is a configuration object that specifies the change logs to monitor for events, or a **`polling`** property, which is a configuration object that identifies resource URLs to monitor for changes.

For backward compatibility reasons, if the **`source`** configuration object contains neither an **`events`** property nor a **`polling`** property, then the properties of the **`source`** configuration object are parsed as an **`events`** configuration, as described below.

##### 3.10.5.1.1 `events`

The **`events`** configuration object contains information about the change log URLs to monitor and authentication for those URLs.

**`urls` (required)**
The **`urls`** property is a list of CUE Content Store change log URLs that CUE Zipline should monitor for events. Each URL in the list identifies a specific change log on the CUE Content Store system.

Alternativlely, you can just add the CUE Content Store web service start URL to the list. In this case, CUE Zipline will monitor **all** change logs accessible with the credentials supplied in the **`auth`** property (see below).

**`types` (optional)**
When the **`urls`** property contains the CUE Content Store web service index page, this property can be used to filter the type of change logs monitored.

The property is a list, where each entry is one of **`publication`**, **`person`**, **`section`**, or **`publication-structural`**.

**`auth` (required)**
The **`auth`** configuration property defines the credentials used for accessing the change logs defined in the **`urls`** list, and is described in section 3.10.5.2.

At the moment, only **`basic`** authentication is supported for monitoring change logs.

**`propagation`**
A set of propagation rules, one for each CUE Content Store **object type** that may be updated. A rule defines:

- Whether or not related objects should be updated together with this object
- If so, which related objects should be updated

Copyright © 2020-2023 Stibo DX A/S

When a CUE Content Store object receives an update event, the external processor uses these rules to find which related objects should also be updated, and passes the update event on to those objects.

The object types that may be specified are represented by the following keywords:

**`article`**
**`container`**
**`inbox`**
**`list`**
**`list-pool`**
**`person`**
**`pool-entry`**
**`section`**
**`section-page`**

The following example specifies that if an **active** section page is updated, the update event will be passed on to the section page's parent (that is, the section it belongs to). This rule thereby ensures that a section is updated whenever its active section page is updated.

```
propagation:
  section-page:
    when:
      state: active
    relations:
      - parent
```

Propagation rules are applied recursively, so in the above case, updating the section page's parent section would also trigger the processing of any **`section`** propagation rules that have been specified. The external processor keeps track of all objects queued for updating and ensures that no objects are updated more than once.

More than one propagation rule may be specified for a given object type (allowing different relations to be updated when different conditions are satisfied).

Each propagation rule may contain the following properties:

**`when` (optional)**
: Defines a set of conditions governing whether or not an update event is to be propagated (passed on to related objects). An event is only propagated if all the specified conditions are satisfied. If no **`when`** property is specified, then the event is always propagated.

  The syntax and semantics of the **`when`** condition are described in section 3.9.

**`relations` (required)**
: The relation types to follow when propagating an update event. If the **`when`** condition is satisfied (or if no **`when`** condition has been specified) then the external processor follows all links of the specified types and passes the targets of the links to the external event handler.

  The list of relation types can be specified either as a string containing a comma-separated list of relation types or as a YAML/JSON array. For instance, the following two examples are identical:

```
relations: home-section, container

relations:
  - home-section
  - container
```

Copyright © 2020-2023 Stibo DX A/S

The following relation type keywords may be specified:

**`changelog`**
**`container`**
**`container-item`**
**`content-items`**
**`home-section`**
**`lock`**
**`model`**
**`organizational-unit`**
**`parent`**
**`publication`**
**`storyline`**

The keywords correspond to selected link relation URLs used in Content Store web service Atom resources (see Supported Relations).

###### 3.10.5.1.2 `polling`

CUE Zipline is able to poll resource URLs at regular intervals and forward events to an external event handler when the contents of the resource change.

Each time a resource URL is resolved, the value of the ETag header in the response is compared to the previous value. If the value has changed, then the external processor is notified.

If the response doesn't contain an "ETag" header, then the resource contents are hashed and the event handler is notified if the hash value changes.

The **`polling`** configuration object has the following properties:

**`urls` (required)**
: A list of resource URLs to poll.

**`interval` (required)**
: Properties defining the interval at which URLs are polled. Each URL listed in **`urls`** is polled at this interval. If there are multiple URLs in the list, then polling of each URL is spread out over the interval, to ensure that all the requests are not sent at the same time.

  The interval properties are described below. Multiple properties can be combined. And at least one non-zero value must be specified.

  **`weeks` (optional)**
  : A floating point number indicating the number of weeks between polling each URL.

  **`days` (optional)**
  : A floating point number indicating the number of days between polling each URL.

  **`hours` (optional)**
  : A floating point number indicating the number of hours between polling each URL.

  **`minutes` (optional)**
  : A floating point number indicating the number of minutes between polling each URL.

  **`seconds` (optional)**
  : A floating point number indicating the number of seconds between polling each URL.

**`auth` (optional)**

The **`auth`** configuration property defines the authentication mechanism to use if authentication is needed for polling the configured URLs.

For URL polling, both **`basic`** and **`oauth`** authentication is possible.

The configuration options for **`auth`** is described in section 3.10.5.2.

**`accept_duplicates` (optional)**

If the **`accept_duplicates`** property is set to "true", then all events for a resource are sent to the event handler. The event handler is then responsible for ignoring duplicates if necessary.

By default, CUE Zipline ignores duplicate events that have already been processed successfully.

For example, a resource that is polled every 2 minutes but only changes every 10 minutes will produce 5 events for the same resource value. If the first event is processed without error, then the four subsequent events are ignored.

If an event for a given resource value results in an error, then subsequent events for that resource are still forwarded to the event handler until one completes without error or the resource value changes.

#### 3.10.5.2 `auth`

The **`auth`** configuration object contains one of a set of supported properties (**`basic`**, **`oauth`**), which in turn contain properties for authentication using that method.

##### 3.10.5.2.1 `basic`

CUE Zipline can provide basic authentication when accessing CUE Content Store change logs or polling URLs. The **`basic`** configuration object contains the properties needed for authenticating.

**`username` (required)**

The user name for accessing the target server.

**`password` or `password_file` (required)**

The password for accessing the target server.

As an alternative to providing the password in the configuration file, CUE Zipline supports reading it from a file.

The **`password_file`** property should contain the path of a file, from which CUE Zipline will read the first line as the password to use.

##### 3.10.5.2.2 `oauth`

CUE Zipline supports the "client credentials" workflow of OAuth2 for authenticating when polling URL resources.

The **`oauth`** configuration object contains the following properties:

**`token_url` (required)**

The **`token_url`** property contains the URL from which CUE Zipline can retrieve an authorization token.

**`client_id` (required)**

The **`client_id`** property contains the client ID to use for authentication.

**`client_secret` or `client_secret_file` (required)**

The **`client_secret`** property contains the client secret to use for authentication.

Copyright © 2020-2023 Stibo DX A/S 

As an alternative to providing the client secret in the configuration file, CUE Zipline supports reading it from a file.

The **client_secret_file** property should then contain the path of a local file on the server running CUE Zipline, from which CUE Zipline will read the first line as the client secret to use.

**audience (required)**

The **audience** property contains a value that identifies the resource CUE Zipline is trying to get access to.

The value is provided as a parameter to the **token_url** and is typically provided by the OAuth2 server along with the client ID and secret.

## 3.11 copyback

The **copyback** property contains configuration parameters for the CUE Zipline copy-back feature that allows CUE Print users to copy small changes and additions made to packages back to their source print storylines, or to create new content through ordering (see section 1.3). The configurations specified here have two parts, **fields** and **create**, which handle updating and creating content respectively.

**fields** only affect the copy-back feature's handling of asset metadata. When a new asset such as an image is added to a package, or an existing asset is changed in some way and the CUE Print user chooses to copy the change back to CUE then the **copyback** property determines what metadata is copied back, and where it is copied to.

Each entry under **create** corresponds to a container name (as configured in CUE Print). Each container name maps onto a container type, and has a set of field mappings both for the container and content to be created. Additionally it is possible to specify the root template that should be used to generate the storyline. The root template must be a jinja file in *config***/conversion-templates/cue-print/cue-print-to-storyline**.

```
copyback:
  # Picture content fields
  fields:
    - name: title
      value:
      - meta: filename
    - name: caption
      value:
      - attribute: CaptionText
      - attribute: IIM_Caption
    - name: byline
      value:
      - attribute: CaptionByline
      - attribute: IIM_Byline
    - name: credit
      value:
      - attribute: CaptionCredit
      - attribute: IIM_Credit
  # Field mappings for when copyback needs to create new content.
  create:
      # Container name as configured on cue-print
    - name: Regular News Story
      # Container type from content store
      type: regular-news-story
```

```
        # root template used for creating storyline
        root_template: storyline.jinja2
        container:
          # Container object fields
          fields:
            - name: com.escenic.container.slug
              value:
              - attribute: Name
        content:
          # Story content fields
          fields:
           - name: title
             value:
             - attribute: Name
```

**copyback** contains a **fields** and a **create** property.

**fields** is an array in which each element defines a mapping between a Content Store field and the CUE Print attributes that can be used to fill it. Each element contains the following properties:

**name (required)**
The name of a content item field. If the target content item has a field with this name, then **value** is used to set it.

**value (required)**
An array of possible sources in the CUE Print asset from which the **name** field can be filled. The sources are tried in order, and the first one that contains a value is used. Two types of source are possible:

**attribute (required)**
The name of a CUE Print attribute.

**meta (required)**
An item of metadata extracted from the asset itself (an image file for example). Currently the only value that may be specified here is **filename**. It means the name of the asset file (name only, no path).

**create** is an array in which each element corresponds to a container type, that can be targeted for creating new content. Each element contains the following properties:

**name (required)**
The name of the container. This is a name configured in CUE Print.

**type (required)**
The container type. This is the unique name of the container resource as defined in the Content Store.

**root_template (optional)**
The Jinja file used as the root template for the storyline. The default value is **storyline.jinja2**.

**container (required)**
A configuration for the container to be created.

**fields (required)**
An array defining field mappings between CUE Print and Content Store fields. It works in exactly the same way as the **fields** array defined under **copyback**.

Copyright © 2020-2023 Stibo DX A/S

**`content` (required)**
A configuration for the content to be created.

**`fields` (required)**
An array defining field mappings between CUE Print and Content Store fields. It works in exactly the same way as the **`fields`** array defined under **`copyback`**.

## 3.12 dcx-converters

**`dcx-converters`** contains a single property, **`wire/target`**, that specifies how a DC-X Wire story is converted into a CUE Content Store story given the publication, container and content-type.

```
dcx-converters:
  wire:
    target:
      publications:
      - text: Tomorrow Online
        value: tomorrow-online
        containers:
        - text: Regular News Story
          value: regular-news-story
          fields:
          - name: Headline
            meta: com.escenic.container.slug
        content-types:
        - text: Storyline
          value: storyline
          fields:
          - name: Headline
            meta: title
          template-uri: dcx/wire/wire-to-storyline/storyline.jinja2
          binary-relation-group: relations
        binary-content-types:
        - text: Picture
          value: picture
          fields:
          - name: ImageCaption
            meta: caption
          - name: _display_title
            meta: title
        content-duplication:
          allowed-for:
          - tomorrow-sport
```

**`containers` (required)**
List of containers inside the given publication that can be the destination of the converted story.

**`fields` (optional)**
List of field mapping between the DC-X story and the container.

**`content-types` (required)**
List of content-types inside the given publication that can be the destination of the converted story.

**`fields` (optional)**
List of field mapping between the DC-X story and the content type.

Copyright © 2020-2023 Stibo DX A/S

**`template-uri` (required)**
Path to template file that maps a DC-X story to a CUE Content Store story

**`binary-relation-group` (optional)**
The relation group to use when linking binary assets to the CUE Content Store story

**`binary-content-types` (optional)**
List of field mapping for binary content that is linked to the destination story.

**`content-duplication` (optional)**
If specified, contains a single **`allowed-for`** property containing a list of publications in which wire stories belonging to this publication may be duplicated. For further information, see section 3.12.1.

### 3.12.1 Content Duplication

By default, CUE Zipline will not re-import a wire story that already exists somewhere in the Content Store. It returns the URI of the existing story to CUE instead.

The content-duplication configuration object lets you modify this behavior and allow duplication of wire stories in some circumstances, by specifying

duplication rules for some or all of your publications as follows:

```
publications:
- text: Tomorrow Online
  value: tomorrow-online
  content-duplication:
    allowed-for:
    - tomorrow-sport
```

The above configuration means that if CUE Zipline has received a request to import a wire story to **`tomorrow-sport`** that has already been imported to the **`tomorrow-online`** publication, then it should create a duplicate. If it is importing the story to any other publication, however, then it should not create a duplicate, and return the URI of the existing **`tomorrow-online`** story to CUE instead.

You can include the names of multiple publications under the **`allowed-for`** property if required, for example:

```
publications:
- text: Tomorrow Online
  value: tomorrow-online
  content-duplication:
    allowed-for:
    - tomorrow-sport
    - living-online
```

You can also use a wild card to allow duplication for all publications except the current one:

```
publications:
- text: Tomorrow Online
  value: tomorrow-online
  content-duplication:
    allowed-for:
    - "*"
```

Copyright © 2020-2023 Stibo DX A/S

This means that if you are importing to any publication except **tomorrow-online** itself, then duplication is allowed. If you want to allow duplication of **tomorrow-online** wire stories in **tomorrow-online** itself, then you must add an explicit entry for that as well:

```
publications:
- text: Tomorrow Online
  value: tomorrow-online
  content-duplication:
    allowed-for:
    - "*"
    - tomorrow-online
```

If duplication is enabled for any publications at an installation, then the Content Store may contain several copies of some wire stories. This means that when CUE Zipline is importing a wire story in a context where duplication is **not** allowed, it may have several existing copies of the wire story to choose from. In such cases, CUE Zipline will choose a copy belonging to the target publication itself, if one exists. Otherwise it will just choose any copy.

## 3.13 audit

The **audit** property is used to configure CUE Zipline's audit trail feature, which writes a record of all actions performed to a log file. This log file is produced specifically for audit purposes and is separate from the diagnostic log produced by the general error logging feature (see section 3.6). **audit** contains a single property, **logging**. Under this is a standard Python logging configuration as described here.

```
audit:
  logging:
    formatters:
      minimal:
        format: '%(asctime)s - %(message)s'
    handlers:
      file:
        class: logging.handlers.RotatingFileHandler
        level: DEBUG
        formatter: minimal
        filename: /var/log/zipline/zipline-audit.log
        maxBytes: 1073741824
        backupCount: 5
        encoding: UTF-8
    root:
      level: INFO
      handlers:
      - file
```

## 3.14 cluster

The **cluster** property is used to configure a CUE Zipline cluster. It describes the members of the cluster and how they communicate. Clustering is optional. If you only intend to run a single instance of CUE Zipline then the **cluster** property can be omitted. If you do intend to run a cluster, then each CUE Zipline instance in the cluster must have a similar (but not identical) **cluster** property definition. In a cluster of two, for example, the instances might have the following cluster definitions:

```
cluster:
  instance_id: zipline01
  instance_name: Zipline 1
  listen_address: 0.0.0.0:12790
  members:
    - zipline1.myproject.com:12790,zipline2.myproject.com:12790
    - zipline1.myproject.com:12790,zipline2.myproject.com:12790
```

and:

```
cluster:
  instance_id: zipline02
  instance_name: Zipline 2
  listen_address: 0.0.0.0:12790
  members:
    - zipline1.myproject.com:12790,zipline2.myproject.com:12790
    - zipline1.myproject.com:12790,zipline2.myproject.com:12790
```

**`instance_id` (optional)**
: The internal ID of this CUE Zipline instance. The ID must be unique within the cluster. If not specified then it is set to the value of the **`ZL_CLUSTER_INSTANCE_ID`** environment variable. If **`ZL_CLUSTER_INSTANCE_ID`** is not set, then it is set to an automatically assigned UUID.

**`instance_name` (optional)**
: A descriptive name for the cluster instance. If not specified then it is set to the value of the **`ZL_CLUSTER_INSTANCE_NAME`** environment variable. If **`ZL_CLUSTER_INSTANCE_NAME`** is not set, then it is set to the name of the host.

**`listen_address` (optional)**
: The network address and port number to listen on for internal communication between CUE Zipline instances. The network address and port number must be accessible to all other instances in the cluster. If not specified then it is set to the value of the **`ZL_CLUSTER_LISTEN_ADDRESS`** environment variable. If **`ZL_CLUSTER_LISTEN_ADDRESS`** is not set, then it is set to **`0.0.0.0:12790`**, which means "listen on port 12790, on all the host's network interfaces".

**`members` (optional)**
: An array containing the network address and port number of each instance in the cluster. If not specified then it is set to the value of the **`ZL_CLUSTER_MEMBERS`** environment variable. If **`ZL_CLUSTER_MEMBERS`** is not set, then it is set to an empty array.

  The value of **`ZL_CLUSTER_MEMBERS`** must be a comma-separated list of entries. For example: **`zipline1.myproject.com:12790,zipline2.myproject.com:12790`**.

  If **`members`** is undefined or left as an empty array, then CUE Zipline will run as a single instance (always active).

## 3.15 monitoring

The **`monitoring`** property is used to configure the monitoring endpoint.

**`default_range` (optional)**
: Defines the default time period for monitoring reports. This default is used for:

  - On-demand reports, where the submitted request does not include a **`period`** parameter (see section 10.1).

- Automated reports written to the CUE Zipline log file.

You may specify a single time period only, specified in either hours (`"24h"`), minutes (`"30m"`), or seconds (`"60s"`).

If no **`default_range`** and no **`period`** request parameter is specified, then the internal default time of 15 minutes is used.

**`log_interval` (optional)**
Defines the interval between the automatically generated status reports that are written to the CUE Zipline log file. The interval can be specified in in either hours (`"24h"`), minutes (`"30m"`), or seconds (`"60s"`). You can disable the automatic generation of status reports by entering `"none"`.

If not specified then it is set to the value of the **`ZL_MONITORING_LOG_INTERVAL`** environment variable. If **`ZL_MONITORING_LOG_INTERVAL`** is not set, then automatic reports are generated every 15 minutes.

## 3.16 CUE Print Asset Mapping

Mapping of assets in CUE Content Store is controlled based on configuration files in the **`/etc/cue/zipline/asset-mapping`** folder.

When mapping a content item, CUE Zipline will look for configuration files in this folder, based on the content type and/or the name of the publication owning the content item.

CUE Zipline will load configuration files from the folder based on a naming scheme. The supported schemes are (in order, from highest to lowest priority):

**<publication-name>.<content-type-name>.config.yaml**
Files with this naming pattern contain mapping configurations for a specific content type in a specific publication. For instance, to configure the mapping for the "storyline" content type in the "tomorrow-online" publication, create a mapping in a file named **`tomorrow-online.storyline.config.yaml`**.

**<publication-name>.config.yaml**
Files with this naming pattern contain mapping configurations for all the content types in a publication. For instance, configurations in a file named **`tomorrow-online.config.yaml`** will apply to all content types in the "tomorrow-online" publication.

**<content-type>.config.yaml**
Files using this naming scheme contain mapping configurations for a specific content type, regardless of which publication it is in. For instance, the file **`storyline.config.yaml`**, contains general configuration settings for content items of the type "storyline" in any publication.

**config.yaml**
This is the most generic mapping configuration. It applies to all content types in all publications, but is overridden by configurations in any of the other more specific files.

Each configuration must contain a single configuration object, defining the following properties:

**`type`**
Defines the type of conversion to use for the content item being processed. The valid values are:

Copyright © 2020-2023 Stibo DX A/S

**`text`**
Maps the content item to CUE Print, using the templates for text conversion in the **`text`** sub-folder of the **`storyline-to-cue-print`** template folder.

**`image`**
Maps the content item to CUE Print by transferring image (or graphics) data and converting the image caption using templates in the **`image`** sub-folder.

**`attributes`**
Defines the mapping of attributes from the CUE Content Store content item to the target CUE Print object. The list of target attributes is currently limited, based on the target object type.

The property must contain a list of mapping definitions, as described in section 3.17.

**`template_vars`**
Defines the mapping of attributes from the CUE Content Store content item to be used in the jinja template for the target CUE Print object. The attributes mapped can be accessed in the jinja template as **`template_vars`**

The **`template_vars`** mapping is currently only available for storylines.

The property must contain a list of mapping definitions, as described in section 3.17.

## 3.17 Mapping Attributes

There are several places in the CUE Zipline configuration where there's a need for mapping attributes from a CUE storyline to fields/attributes of an external system.

These mappings share a common format, which is a list of definitions, each providing the target field (or attribute) name and the source of the field contents. The target definition contains just the name of the target field or attribute:

**`name` (required)**
Defines the name of the target attribute or field.

The source definition is identified by one of the following properties, possibly with additional properties, as described below.

**`context` (optional)**
The value of the **`context`** property is used to define the source object that should be used when retrieving the value.

Possible context values:

**authors**
Every author in the story. Result will be an array of values.

**profiles**
Every author profile of the person. Specifically the relations marked with the **`metadata:group="com.escenic.profiles"`** attribute. Result will be an array of values.

**container**
The container of the story.

**creator**
The person that created the story.

Copyright © 2020-2023 Stibo DX A/S 

**home-section**
The home section in the story.

**operator**
The person that last changed the story.

**publication**
The publication of the story.

**sections**
Every section in the story. Result will be an array of values.

**`feature` (optional)**
The value of the **`feature`** property is used as the name of a publication feature, the value of which is used as the contents of the target field/attribute.

The **`feature`** property is only available in attribute mappings for internal processors. Not for mappings defined in external event processors.

**`field` (optional)**
If a definition entry contains a **`field`** property, but no **`type`** property, then it's interpreted as a mapping from a content item field.

The value of the content item field is extracted as a string and used as the value of the target field/attribute.

**`meta` (optional)**
Maps a meta-data attribute of the object being processed, as described in the section section 3.17.1.

**`type` (optional)**
The **`type`** property identifies the first element of a given type in a storyline. The required **`field`** property, in the same mapping definition, then identifies a field in that element. The value of that field is then used as the value of the mapping target.

For instance, the following mapping definition, extracts the value of the "caption" field in the first "image" storyline element as the value of the "caption" target attribute:

```
- name: caption
  type: image
  field: caption
```

**`summary-field` (optional)**
Sets the target attribute to the value of the summary field of a relation with the field name matching the value of this property.

**`template` (optional)**
Renders the target field contents from a template provided inline in the configuration. See section 3.17.2 below.

**`template-uri` (optional)**
Like **`template`**, but the template is read from a file.

**`first-of` (optional)**
Defines a list of possible values, trying each one, in order, until a non-empty value is found. The value of the property is a list of source definitions.

For example, the following mapping will fill the "target" field with the contents of the "caption" field of the "image" storyline element if non-empty and otherwise use the "caption" field of the content item:

```
- name: target
```

Copyright © 2020-2023 Stibo DX A/S

```
first-of:
- type: image
  field: caption
- field: caption
```

For backward compatibility reasons, **`element`** can be used as the property name as well.

**`static`** **(optional)**

Sets the target attribute to the configured value. For instance the following configuration will set the field called **`target`** to the value **`"Approved"`**.

```
- name: target
  static: Approved
```

**`object`** **(optional)**

Sets the target attribute to an indexed array of multiple source values. It should contain a list of source attribute definitions. The following configuration will set two properties in the target.

"value" to the content of the story creator's title property

"email" to the content of the story creator's email field.

```
- name: target
  object:
    - name: value
      meta: title
    - name: email
      field: com.escenic.emailAddress
  context: creator
```

In addition to the target and source properties, the mapping definition may contain a **`mapping`** property, defining a simple lookup table, changing the source value into something else.

**`mapping`** **(optional)**

The **`mapping`** property contains a set of "from" and "to" values, in the form of a dictionary.

For instance, the following mapping will change the state "writing" to "ToWriting", "copy-editing" to "ToCopyEditing", and "ready" to "Ready" befory applying the value to the "WorkFlowGroupName" target attribute:

```
- name: WorkFlowMailGroupName
  meta: state
  mapping:
    writing: ToWriting
    copy-editing: ToCopyEditing
    ready: Ready
```

### 3.17.1 Meta Attributes

Meta attributes identifies information about the content item that are not contained in fields or as part of the storyline. The possible values are:

**`author`**

Extracts the full name of the first author listed in the content item and uses that as the value of the target field.

**`authors`**

Extracts the full name of all authors listed in the content item and supplies that as a list of values for the target field.

Copyright © 2020-2023 Stibo DX A/S

**`container-url`**
The container's Content Store URL. Resolving this URL will provide a XML structure describing the container object.

**`content-url`**
Identifies the web-site URL of a published content item (or section). There's no value if the content item isn't published and, in this case, the field/attribute will not be set on the target object.

**`cook-url`**
Resolves to the updated item's Cook URL. Resolving this URL will provide a JSON structure describing the content object from CUE Front's Cook.

Mapping of this attribute requires configuring a **`dpres_cook`** endpoint defining the base URL of the CUE Front Cook, as described in section 3.2.

**`creation-date`**
Extracts the creation date of the content item and uses that as the value of the target field.

**`creation-date`**
Extracts the creation date of the content item and uses that as the value of the target field.

**`creator`**
Extracts the full name of the user listed as the creator in the content item and uses that as the value of the target field.

**`home-section`**
Extracts the full name of the home section in the content item and uses that as the value of the target field.

**`home-section-url`**
The home section's Content Store URL. Resolving this URL will provide an XML structure describing the home section object.

**`id`**
Uses the CUE Content Store id (resource URL) of the content item as the value of the target attribute.

**`local-origin-url`**
Identifies the local base variant of the content item and uses its web-site URL as the value of the mapping target.

The local base variant is identified as the first variant in the same organizational unit as the content item being mapped.

If the content item being mapped **is** the local base variant, then this is identical to a mapping of the **`content-url`**.

The **`local-origin-url`** property is only available in attribute mappings for internal processors. Not for mappings defined in external event processors.

**`operator`**
Extracts the full name of the user listed as **`last-edited-by`** in the content item and uses that as the value of the target field.

**`origin-url`**
Identifies the base variant of the content item and uses its web-site URL as the value of the mapping target.

The base variant is identified as the first variant in the same container as the content item being mapped.

Copyright © 2020-2023 Stibo DX A/S 

If the content item being mapped **is** the base variant, then this is identical to a mapping of the **`content-url`**.

The **`origin-url`** property is only available in attribute mappings for internal processors. Not for mappings defined in external event processors.

**`ou/organizational-unit`**
Extracts the name of the content item's organizational unit and uses that as the target value.

**`publication`**
Uses the name of the content item's owner publication as the value of the target field.

**`publish-date`**
Extracts the publication date of the content item and uses that as the value of the target field.

**`sections`**
Extracts the name of all sections listed in the content item and supplies that as a list of values for the target field.

**`source`**
Uses the source attribute of the content item as the value of the target attribute.

**`source-id`**
Uses the source id attribute of the content item as the value of the target attribute.

**`state`**
Uses the workflow state of the content item as the value of the target attribute.

**`summary`**
Extracts the value of the content item's summary attribute and uses that as the target value.

**`title`**
Extracts the value of the content item's title attribute and uses that as the target value.

**`type`**
The **`type`** meta-data attribute will set the target field/attribute to a string containing an object type name.

The possible object type names are:

**`article`**
**`container`**
**`inbox`**
**`list`**
**`list-pool`**
**`person`**
**`pool-entry`**
**`section`**
**`section-page`**

**`storyline`**
The **`storyline`** mapping is a special mapping useful in the context of a template mapping definition. It extracts the storyline into the context of the template. It's not useful as a general mapping.

### 3.17.2 Templates

A target field or attribute can be filled with the result of rendering a template. As all templates used in CUE Zipline, field mapping templates are based on Jinja2.

The template can either be provided inline, using the **template** property, or in a separate file, using the **template-uri** property.

The template is executed with a context that defines the variables available to the template. The context is defined using a **template-vars** property, which contains a list of mapping definitions.

For instance, the following mapping will render the contents of a "byline" target attribute from a template using the author name as a context variable:

```
- name: byline
  template: "By {{ author_name }}"
  template-vars:
  - name: author_name
    meta: author
```

Templates can be loaded from a file, instead of being provided inline. For this, use the **template-uri** property and set it to the path of the template file.

## 3.18 locations

The **locations** object provides a convenient mechanism for including parts of the CUE Zipline configuration from files stored in other locations. Currently it may contain only one property, **processors**:

```
locations:
  processors: ./processors.d
```

The **processors** property must be set to the path of a folder containing one or more **.yaml** files. The path must be specified relative to the location of the configuration file. Each **.yaml** file in the folder must contain the definition of one or more processors (as defined in section 3.10). Files that contain more than one processor definition must separate the definitions using the **.yaml ---** syntax – that is, each processor definition must be preceded by a line containing three hyphens:

```
---
```

The primary purpose of the locations property is to simplify the installation of CUE Zipline plugins. When a plug-in is installed, it adds sample processor configurations to the **/etc/cue/zipline/processors.d** folder, where they will be automatically detected and loaded the next time CUE Zipline is loaded. This means that installing a plugin generally involves only running the install command and then editing the configuration file added to the **/etc/cue/zipline/processors.d** folder.

## 3.19 Environment Variables

This section describes a number of environment variables that can affect the configuration. For some settings, this can be used as an alternative to customising a common configuration file.

Some settings are only configurable using environment variables, as they are not expected to need configuration under normal circumstances.

Copyright © 2020-2023 Stibo DX A/S 

At startup, CUE Zipline will search the working directory for a file named **zipline.env** and load any environment variables from that file that are not already defined in the system environment.

If no such file is found in the working directory, the application will search successive parent directories for the file, until the root directory has been tried.

If no **zipline.env** file is found, CUE Zipline tries looking for a file named **.env**, using the same strategy.

**CA_BUNDLE_PATH**
**CURL_CA_BUNDLE**
: The path of a certificate bundle that includes any custom CA certificates used to generate self-signed certificates for TLS.

  For more information, see section 2.5.

**ZL_CLUSTER_INSTANCE_ID**
: An internal ID of this instance in the Zipline cluster. If a value is configured in the **cluster.instance_id** property of the CUE Zipline configuration file, as described in section 3.14, then it supersedes the value of this environment variable.

**ZL_CLUSTER_INSTANCE_NAME**
: A descriptive name for this instance in the Zipline cluster. If a value is configured in the **cluster.instance_name** property of the CUE Zipline configuration file, as described in section 3.14, then it supersedes the value of this environment variable.

**ZL_CLUSTER_LISTEN_ADDRESS**
: The network address used for internal communication between instances in the cluster. If a value is configured in the **cluster.listen_address** property of the CUE Zipline configuration file, as described in section 3.14, then it supersedes the value of this environment variable.

**ZL_CLUSTER_MEMBERS**
: A comma-separated list of network addresses of cluster members. If a value is configured in the **cluster.members** property of the CUE Zipline configuration file, as described in section 3.14, then it supersedes the value of this environment variable.

**ZL_SERVER_ADDRESS**
: The network interface where CUE Zipline will listen for requests from the CUE editor. If a value is configured in the **server.address** property of the CUE Zipline configuration file, as described in section 3.4, then it supersedes the value of this environment variable.

**ZL_SERVER_PORT**
: The network port where CUE Zipline will listen for requests from the CUE editor. If a value is configured in the **server.port** property of the CUE Zipline configuration file, as described in section 3.4, then it supersedes the value of this environment variable.

**ZL_SERVER_CONTEXT_PATH**
: The application context path used for CUE Zipline. If a value is configured in the **server.context-path** property of the CUE Zipline configuration file, as described in section 3.4, then it supersedes the value of this environment variable.

**ZL_EVENT_LISTENER_PROGRESS_PATH**
: The path of a file used to hold progess state for CUE Zipline between restarts. If a value is configured in the **event_listener.progress_path** property of the CUE Zipline configuration file, as described in section 3.3, then it supersedes the value of this environment variable.

Copyright © 2020-2023 Stibo DX A/S

**ZL_EVENT_HANDLING_MAX_DELAY**

The allowed interval (in seconds) between the receipt of an event notification from the CUE Content Store and the processing of that event by CUE Zipline.

If it takes longer than this, CUE Zipline will emit a warning in the log file.

The default value is 20 seconds.

**ZL_CHANGE_LOG_SECS_BETWEEN_READS**

The minimum number of seconds between reading a CUE Zipline change-log to check for updates. If update notifications are received faster than this, processing of those notifications will be delayed until the configured number of seconds have passed since last reading the change-log.

The default value is 1 (one) second.

**ZL_CHANGE_LOG_POLL_INTERVAL**

The minimum number of seconds between CUE Zipline polling the CUE Zipline change-logs for updates, if no update notifications have been received.

In general, polling should not be necessary because CUE Zipline receives update notifications from CUE Zipline using SSE, so this property should be set to a very high value.

The default value is 600 seconds (that is, 10 minutes).

**ZL_CUE_CS_EXTERNAL_SYSTEM_ID**

A common system name for the CUE Zipline system, used to differentiate external IDs of content when connecting multiple CUE Zipline systems to one CUE Print or DC-X system.

If a value for this system name is configured in the **endpoints.content_store.system-id** property of the CUE Zipline configuration file, then it supersedes the value of this environment variable.

**ZL_CUE_PRINT_EXTERNAL_SYSTEM_ID**

A custom system name for the CUE Zipline system, used to differentiate external IDs of content when connecting multiple CUE Zipline systems to one CUE Print system.

If a value for this system name is configured in the **system-id** property of a CUE Print processor in the CUE Zipline configuration file, then it supersedes the value of this environment variable.

**ZL_DCX_EXTERNAL_SYSTEM_ID**

A custom system name for the CUE Zipline system, used to differentiate external IDs of content when connecting multiple CUE Zipline systems to one DC-X system.

If a value for this system name is configured in the **system-id** property of a DC-X related processor in the CUE Zipline configuration file, then it supersedes the value of this environment variable.

**ZL_MONITORING_LOG_INTERVAL**

The interval between automatically generates status reports. If a value is configured in the **monitoring.log_interval** property of the CUE Zipline configuration file, as described in section 3.15, then it supersedes the value of this environment variable.

**ZL_AUDIT_LOG_GRACE_PERIOD**

Defines the number of seconds the active instance in a cluster keeps internal events around after they have been distributed to inactive instances.

This allows instances to be away for this many seconds (e.g., during a restart) without the need to apply the entire state of the audit log when the instance re-appears.

The default value is 300 (that is, 5 minutes).

**`ZL_AUDIT_LOG_TIMEOUT_ACK`**

The number of seconds to wait for a passive instance to acknowledge events sent to it. Increase this number only if the latency between instances is very large, causing a round-trip to last more than the default value.

The default value is 1.0 (that is, 1 second).

**`ZL_AUDIT_LOG_TIMEOUT_HEARTBEAT`**

The number of seconds between sending heartbeats to passive instances.

By default, the active instance will regularly send a heartbeat to each passive instance to propagate the internal events across the cluster.

If this timeout value is increased, then the maximum delay between registering events on the active instance and the passive instances is increased as well.

The default value is 2.0 (that is, 2 seconds).

Copyright © 2020-2023 Stibo DX A/S

# 4 Conversion Templates

CUE Content Store, CUE Print and DC-X are all highly flexible system that allow documents and data structures to be customized in various ways. CUE Zipline makes frequent use of templates in order to be able to deal with this flexibility – most of the data transformations performed by CUE Zipline use templates to help generate correctly formatted output. A set of standard templates are supplied with CUE Zipline. These will work at many installations, but they may not produce exactly the desired results: they may not, for example include custom fields in converted content. In other cases, the supplied templates may not work at all.

In most cases, some template modifications will need to be carried out to produce the desired results.

CUE Zipline uses the Jinja2 template processor. The Jinja2 templates are all intended to export CUE storylines to some external target format, or else to import from some external format into CUE. All the external target/source formats are XML formats of one kind or another, with a hierarchical internal structure. The storyline data that CUE Zipline retrieves from the Content Store, on the other hand is JSON data, and has a more or less flat internal structure. All the story elements in the storyline are listed in a single **`storyElements`** array, and the relationships between them are specified indirectly (see section 4.1.1 for an example).

Converting between this kind of flat data structure and the hierarchical external structures is not easy to do using a templating language such as Jinja2. In order to simplify the process, therefore, CUE Zipline provides a built-in converter that converts between the internal storyline format and an intermediate format called the **pseudo-storyline** format. A pseudo-storyline is still a JSON data structure, but it has a hierarchical structure similar to the various external formats. The supplied templates are therefore designed to convert between the pseudo-storyline format and various external formats.

All the templates are located in the **`/etc/conf/conversion-templates`** folder by default. This folder contains the following template subfolders:

**`cue-print/storyline-to-cue-print`**
: This folder contains templates for converting CUE pseudo-storylines into CUE Print texts. These templates are used by the section 1.1 conversion and for generating print previews (see section 1.5). You may need to modify them to achieve the desired results with your content types.

**`cue-print/cue-print-to-storyline`**
: This folder contains templates for converting CUE Print texts into CUE pseudo-storylines. These templates are used by the section 1.3 conversion. You may need to modify them to achieve the desired results with your content types.

**`newsmlg2/storyline-to-newsmlg2`**
: This folder contains templates for converting CUE pseudo-storylines into NewsML files. You may need to modify them to achieve the desired results with your content types.

**`newsmlg2/newsmlg2-to-storyline`**
: This folder contains templates for converting NewsML files into CUE pseudo-storylines. You may need to modify them to achieve the desired results with your content types.

**`classic`**
: This folder contains templates for converting classic (rich text based) content items into CUE pseudo-storylines (**`classic-to-storyline`**) and vice-versa (**`storyline-to-classic`**).

Copyright © 2020-2023 Stibo DX A/S 

These templates are used by the section 1.6. You can both modify these standard templates and/ or add additional sets of templates to be used for converting different types of content.

**`dcx/wire/wire-to-storyline`**
This folder contains templates for converting DC-X stories into CUE pseudo-storylines. You may need to modify them to achieve the desired results with your content types.

## 4.1 A Templating Example

This section provides a detailed description of the various components involved in exporting a simple test storyline to CUE Print. Following this example should help to give you a general understanding of how the conversion process works, enabling you to extend and modify the supplied templates, and create templates of your own. It is assumed that you have a general understanding of how templating systems work, and a basic acquaintance with Jinja2.

Our example storyline looks like this in CUE:

My Bullet Test

Here is an introductory **paragraph**, followed by a list:

- Item one
- Item two
    - Nested one
    - Nested two
- Item three

Concluding paragraph.

It consists, then of a headline, followed by a paragraph, a bulleted list and a second paragraph. The first paragraph contains some bold (that is, annotated) text and the bulleted list contains a sublist, also bulleted.

### 4.1.1 The Storyline JSON Structure

When CUE Zipline retrieves the print variant of this storyline from the Content Store web service, it is supplied in the form of a JSON structure like this:

```
{
  "storyline": {
    "id": "5b3251f0-33ba-11eb-b23d-5b6ae38640b0",
    "version": "1",
    "sourceId": "c9fb7ff2-6e3c-42c0-8c79-40c817bd90f4",
    "model": "https://my-content-store/webservice/escenic/shared/model/storyline-
template/print",
    "inheritedFrom": "https://ece-cue-unstable-nightly.cci.cue.cloud/webservice/
escenic/storyline/ad9c82a8-3314-11eb-b23d-5b6ae38640b0",
    "elements": ["/storyElements/1","/storyElements/2","/storyElements/3","/
storyElements/4","/storyElements/5"]
  },
  "storyElements": {
    "1": {
      "model": "https://my-content-store/webservice/escenic/shared/model/story-
element-type/print_head",
      "fields": [],
```

```
      "elements": ["/storyElements/6"]
    },
    "2": {
      "model": "https://my-content-store/webservice/escenic/shared/model/story-
element-type/print_head_deck",
      "fields": []
    },
    "3": {
      "model": "https://my-content-store/webservice/escenic/shared/model/story-
element-type/print_body",
      "fields": [],
      "elements": ["/storyElements/7","/storyElements/8","/storyElements/9"]
    },
    "4": {
      "model": "https://my-content-store/webservice/escenic/shared/model/story-
element-type/print_assets",
      "fields": []
    },
    "5": {
      "model": "https://my-content-store/webservice/escenic/shared/model/story-
element-type/print_quote",
      "fields": []
    },
    "6": {
      "model": "https://my-content-store/webservice/escenic/shared/model/story-
element-type/headline",
      "fields": [
        {
          "name": "headline",
          "value": "My Bullet Test",
          "annotations": []
        }
      ]
    },
    "7": {
      "model": "https://my-content-store/webservice/escenic/shared/model/story-
element-type/paragraph",
      "fields": [
        {
          "name": "paragraph",
          "value": "Here is an introductory paragraph, followed by a list:",
          "annotations": [
            {
              "index": 24,
              "length": 9,
              "name": "bold",
              "value": true
            }
          ]
        }
      ]
    },
    "8": {
      "model": "https://my-content-store/webservice/escenic/shared/model/story-
element-type/list_bulleted",
      "fields": [],
      "elements": ["/storyElements/10","/storyElements/11","/storyElements/12","/
storyElements/13"]
    },
    "9": {
```


Copyright © 2020-2023 Stibo DX A/S 

```
      "model": "https://my-content-store/webservice/escenic/shared/model/story-
element-type/paragraph",
      "fields": [
        {
          "name": "paragraph",
          "value": "Concluding paragraph.",
          "annotations": []
        }
      ]
    },
    "10": {
      "model": "https://my-content-store/webservice/escenic/shared/model/story-
element-type/paragraph",
      "fields": [
        {
          "name": "paragraph",
          "value": "Item one",
          "annotations": []
        }
      ]
    },
    "11": {
      "model": "https://my-content-store/webservice/escenic/shared/model/story-
element-type/paragraph",
      "fields": [
        {
          "name": "paragraph",
          "value": "Item two",
          "annotations": []
        }
      ]
    },
    "12": {
      "model": "https://my-content-store/webservice/escenic/shared/model/story-
element-type/list_bulleted",
      "fields": [],
      "elements": ["/storyElements/14","/storyElements/15"]
    },
    "13": {
      "model": "https://my-content-store/webservice/escenic/shared/model/story-
element-type/paragraph",
      "fields": [
        {
          "name": "paragraph",
          "value": "Item three",
          "annotations": []
        }
      ]
    },
    "14": {
      "model": "https://my-content-store/webservice/escenic/shared/model/story-
element-type/paragraph",
      "fields": [
        {
          "name": "paragraph",
          "value": "Nested one",
          "annotations": []
        }
      ]
    },
```


Copyright © 2020-2023 Stibo DX A/S

```
    "15": {
      "model": "https://my-content-store/webservice/escenic/shared/model/story-
element-type/paragraph",
      "fields": [
        {
          "name": "paragraph",
          "value": "Nested two",
          "annotations": []
        }
      ]
    }
  }
}
```


The above structure has been simplified to improve legibility: the actual data returned by the web service will contain some additional URLs, IDs and so on that are not relevant for our purposes.

### 4.1.2 The Target CUE Print Structure

In order to be able to import the storyline into CUE Print, the storyline JSON data needs to be transformed into an XML document that looks like this:

```
<cci:ccitext xmlns:cci="urn:schemas-ccieurope.com" xmlns:ccix="http://
www.ccieurope.com/xmlns/ccimlextensions">
  <cci:head>
    <cci:p>My Bullet Test</cci:p>
  </cci:head>
  <cci:head_deck/>
  <cci:body>
    <cci:p>Here is an introductory <cci:bold>paragraph</cci:bold>, followed by a
 list:</cci:p>
    <cci:bullet_list>
      <cci:p>Item one</cci:p>
      <cci:p>Item two</cci:p>
      <cci:bullet_list>
        <cci:p>Nested one</cci:p>
        <cci:p>Nested two</cci:p>
      </cci:bullet_list>
      <cci:p>Item three</cci:p>
    </cci:bullet_list>
    <cci:p>Concluding paragraph.</cci:p>
  </cci:body>
  <cci:byline>
    <cci:p/>
  </cci:byline>
  <cci:quote>
    <cci:p/>
  </cci:quote>
</cci:ccitext>
```

### 4.1.3 The Pseudo-Storyline Structure

As you can see, the JSON data supplied by the Content Store (see section 4.1.1) has a flatter structure than the XML document needed for export to CUE Print (see section 4.1.2). The bolded word in the first paragraph is not nested inside the paragraph, and list items are not nested inside lists. In this JSON structure, formatting and other kinds of annotations are defined by specifying the character

ranges to which they apply, and nested elements are unpacked and referenced indirectly. In line 8, for example, the storyline's top-level elements are referenced as follows:

```
    "elements": ["/storyElements/1","/storyElements/2","/storyElements/3","/
storyElements/4","/storyElements/5"]
```

Further down the file, element 8 (the outer bulleted list) contains a similar entry referencing its contents:

```
      "elements": ["/storyElements/10","/storyElements/11","/storyElements/12","/
storyElements/13"]
```

So in order to simplify the conversion task, CUE Zipline automatically generates the following **pseudo-storyline**, a restructured version of the JSON data that more closely resembles the required XML target structure:

```
{
  "storyline_id": "5b3251f0-33ba-11eb-b23d-5b6ae38640b0",
  "type": "print",
  "elements": [
    {
      "type": "print_head",
      "fields": {},
      "elements": [
        {
          "type": "headline",
          "fields": {
            "headline": {
              "value": "My Bullet Test",
              "ops": [
                {
                  "index": 0,
                  "length": 14,
                  "name": "",
                  "value": true,
                  "text": "My Bullet Test"
                }
              ]
            }
          },
          "elements": [],
          "is_empty": false
        }
      ],
      "is_empty": false
    },
    {
      "type": "print_head_deck",
      "fields": {},
      "elements": [],
      "is_empty": true
    },
    {
      "type": "print_body",
      "fields": {},
      "elements": [
        {
          "type": "paragraph",
          "fields": {
```

Copyright © 2020-2023 Stibo DX A/S 

```
        "paragraph": {
          "value": "Here is an introductory paragraph, followed by a list:",
          "ops": [
            {
              "index": 0,
              "length": 24,
              "name": "",
              "value": true,
              "text": "Here is an introductory "
            },
            {
              "index": 24,
              "length": 9,
              "name": "bold",
              "value": true,
              "sub": [
                {
                  "index": 0,
                  "length": 9,
                  "name": "",
                  "value": true,
                  "text": "paragraph"
                }
              ]
            },
            {
              "index": 33,
              "length": 21,
              "name": "",
              "value": true,
              "text": ", followed by a list:"
            }
          ]
        }
      },
      "elements": [],
      "is_empty": false
    },
    {
      "type": "list_bulleted",
      "fields": {},
      "elements": [
        {
          "type": "paragraph",
          "fields": {
            "paragraph": {
              "value": "Item one",
              "ops": [
                {
                  "index": 0,
                  "length": 8,
                  "name": "",
                  "value": true,
                  "text": "Item one"
                }
              ]
            }
          },
          "elements": [],
          "is_empty": false
```

```
},
{
  "type": "paragraph",
  "fields": {
    "paragraph": {
      "value": "Item two",
      "ops": [
        {
          "index": 0,
          "length": 8,
          "name": "",
          "value": true,
          "text": "Item two"
        }
      ]
    }
  },
  "elements": [],
  "is_empty": false
},
{
  "type": "list_bulleted",
  "fields": {},
  "elements": [
    {
      "type": "paragraph",
      "fields": {
        "paragraph": {
          "value": "Nested one",
          "ops": [
            {
              "index": 0,
              "length": 10,
              "name": "",
              "value": true,
              "text": "Nested one"
            }
          ]
        }
      },
      "elements": [],
      "is_empty": false
    },
    {
      "type": "paragraph",
      "fields": {
        "paragraph": {
          "value": "Nested two",
          "ops": [
            {
              "index": 0,
              "length": 10,
              "name": "",
              "value": true,
              "text": "Nested two"
            }
          ]
        }
      },
      "elements": [],
```

Copyright © 2020-2023 Stibo DX A/S

```
              "is_empty": false
            }
          ],
          "is_empty": false
        },
        {
          "type": "paragraph",
          "fields": {
            "paragraph": {
              "value": "Item three",
              "ops": [
                {
                  "index": 0,
                  "length": 10,
                  "name": "",
                  "value": true,
                  "text": "Item three"
                }
              ]
            }
          },
          "elements": [],
          "is_empty": false
        }
      ],
      "is_empty": false
    },
    {
      "type": "paragraph",
      "fields": {
        "paragraph": {
          "value": "Concluding paragraph.",
          "ops": [
            {
              "index": 0,
              "length": 21,
              "name": "",
              "value": true,
              "text": "Concluding paragraph."
            }
          ]
        }
      },
      "elements": [],
      "is_empty": false
    }
  ],
  "is_empty": false
},
{
  "type": "print_assets",
  "fields": {},
  "elements": [],
  "is_empty": true
},
{
  "type": "print_quote",
  "fields": {},
  "elements": [],
  "is_empty": true
```

Copyright © 2020-2023 Stibo DX A/S

```
      }
    ],
    "inherited_from": "http://my-content-store:8080/webservice/escenic/storyline/
ad9c82a8-3314-11eb-b23d-5b6ae38640b0"
  }
```

### 4.1.4 The Conversion Templates

This pseudo-storyline is then passed through the templates in the **`cue-print/storyline-to-cue-print`** folder in order to produce the required output. The starting point is the **`cue-print/storyline-to-cue-print/ccitext.xml`** file, which contains the skeleton of a CUE Print text:

```
<cci:ccitext xmlns:cci="urn:schemas-ccieurope.com"
             xmlns:ccix="http://www.ccieurope.com/xmlns/ccimlextensions">
    ...
    <cci:head>
        ...
    </cci:head>
    <cci:head_deck>
        ...
    </cci:head_deck>
    <cci:body>
        ...
    </cci:body>
    <cci:byline>
        <cci:p>
            ...
        </cci:p>
    </cci:byline>
    <cci:quote>
        <cci:p>
            ...
        </cci:p>
        ...
    </cci:quote>
    ...
</cci:ccitext>
```

where the `...` ellipses represent Jinja2 template code that extracts content from the pseudo-storyline and inserts it into the CUE Print text. The **`cci:head`** section of the template, for example, actually looks like this:

```
    <cci:head>
        {% for print_head in storyline.elements|of_type('print_head') %}
          {% for element in print_head.elements %}
            {% if element.type == 'headline' %}
        <cci:p>{{element.fields.headline.value}}</cci:p>
            {% elif element.type == 'lead_text' %}
        <cci:p>{{element.fields['lead-text'].value}}</cci:p>
            {% endif %}
          {% endfor %}
        {% endfor %}
    </cci:head>
```

This template code searches the pseudo-storyline's **`elements`** array looking for entries with a **`type`** property set to **`print_head`**. It then picks from this group's **`elements`** array any entries with **`type`** properties of **`headline`** or **`lead_text`** and insert their values, wrapped in **`cci:p`** elements. The storyline in this case only contains a **`headline`**, so the resulting output is:

Copyright © 2020-2023 Stibo DX A/S

```
    <cci:head>
      <cci:p>My Bullet Test</cci:p>
    </cci:head>
```

The **body** section of **cue-print/storyline-to-cue-print/ccitext.xml** includes references to other templates that deal with the various story element types that may appear in the body of the storyline:

```
      <cci:body>
          {%- for print_body in storyline.elements|of_type('print_body') %}
            {%- for element in print_body.elements|
  of_type('headline','lead_text','paragraph','interview', 'list_bulleted',
   'list_numbered') %}
              {% include ['body/' + element.type + '.xml',
                          element.type + '.xml'] %}
            {% endfor %}
          {% endfor %}
      </cci:body>
```

You can therefore easily extend CUE Zipline to support new story element types by adding your own templates to the **cue-print/storyline-to-cue-print/body** folder, and adding a reference here. If, for example, your publications include story elements called **aside**, you can extend this transformation to handle them by adding a suitable **aside.xml** template to the **cue-print/storyline-to-cue-print/body** folder, and adding a corresponding reference to **cue-print/storyline-to-cue-print/ccitext.xml**:

```
            {%- for element in print_body.elements|
  of_type('headline','lead_text','paragraph','interview', 'list_bulleted',
   'list_numbered', 'aside') %}
```

## 4.2 Handling Other Conversions

All the conversions work in the same basic way. In the case of the import templates such as **cue-print/cue-print-to-storyline/storyline.jinja2**, the objective is to output a pseudo-storyline, which CUE Zipline will then convert to the flat storyline format required by the Content Store.

## 4.3 How To

This section contains examples of how template customizations can be used to solve various problems.

### 4.3.1 Supporting Annotated Image Captions

The default templates supplied with CUE Zipline only support plain text image captions. You can, however, upgrade the templates to support formatted (that is, annotated) captions.

The first task is to modify your **image** story element type, if necessary, and ensure that its **caption** field supports the annotations you require (bold and italic, for example). For general information about story element types and how to define them, see Story Element Types..

Copyright © 2020-2023 Stibo DX A/S 

Once that is done, you can then modify your CUE Zipline templates to support the transfer of annotated captions back and forth between Content Store and CUE Print.

#### 4.3.1.1 Content Store to CUE Print

The template **/storyline-to-cue-print/image/ccitext.xml** converts pseudo-storyline versions of Content Store image story elements to CUE Print image caption texts.

The part of the template that is responsible for converting the caption looks like this:

```
<cci:cutline_c>
  {% if storyline.elements and storyline.elements[0].fields.caption.value %}
    {{ storyline.elements[0].fields.caption.value }}
  {% elif summary and summary.fields.caption.value %}
    {{ summary.fields.caption.value }}
  {% else %}
    {{ content.fields.caption.value }}
  {% endif %}
</cci:cutline_c>
```

The first **if** statement specifies what to do if the image storyline element contains a caption, and that is where we want to support annotations. Replace the highlighted line with the following:

```
{% with field=storyline.elements[0].fields.caption %}
  {% include "common/text-content.xml.j2" %}
{% endwith %}
```

This assigns the content of the **caption** field (that is, a JSON object containing both text and markup operations) to a variable called **field** and passes it to an included template, **common/text-content.xml.j2**.

The next task is to create **common/text-content.xml.j2**. Create a **common** sub-folder in the **image** folder and then create **text-content.xml.j2** in the new folder.

Open the new file in an editor the file and enter the following:

```
{% for op in field.ops recursive %}
    {% if op.name == "bold"  %}
        <cci:bold>{{ loop(op.sub) }}</cci:bold>
    {% elif op.name == "" %}
        {{ op.text }}
    {% else %}
        {{ loop(op.sub) }}
    {% endif %}
{% endfor %}
```

The first line (**{% for op in field.ops recursive %}**) starts a loop over the text "operations" in the field. Remember that before executing the conversion templates, CUE Zipline converts the flat storyline to a hierarchical structure. In this specific case, the field's **annotations** property has been converted to a hierarchy of operations (**ops**).

Initially, the line enumerates the top level list of text operations, but the **recursive** keyword specifies that the loop can be executed recursively to process annotations within annotations if necessary (a **bold** annotation within an **italic** annotation, for example).

Copyright © 2020-2023 Stibo DX A/S 

In the next line, the **`if`** statement tests whether the operation name is **`bold`**, in which case a CUE Print bold tag (**`<cci:bold>`**) is wrapped around the operation, which is recursively resubmitted to the loop.

The next part of the **`if`** statement handles the actual text content:

```
{% elif op.name == "" %}
    {{ op.text }}
```

An operation with no name indicates plain text, so the template just outputs the text directly, using **`{{ op.text }}`**.

Finally, the last part of the **`if`** statement handles any annotations that are not to be converted to CUE Print tags:

```
{% else %}
    {{ loop(op.sub) }}
{% endif %}
```

Since the operation isn't plain text, the template just resubmits it to the loop, thereby ensuring that any sub-operations are handled correctly. If, for example, the caption field supports **`italic`** annotations as well as **`bold`**, then any **`italic`** operations will be "caught" by this else section. The **`italic`** operation will be ignored, and its content passed back into the loop for processing. If the content is just plain text, then it will be caught by the **`op.name == ""`** test. If the content includes any **`bold`** operations, then they will be caught and handled by the **`op.name == "bold"`** test, and so on.

If you actually want to convert **`italic`** annotations and convert them to CUE Print tags, then you can do so by adding a test to the template:

```
{% elif op.name == "italic"  %}
    <cci:italic>{{ loop(op.sub) }}</cci:italic>
```

In this way it is possible to catch all annotations supported by the source story element type and either convert them to corresponding tags in the target CUE Print text or ignore them and just pass on the content.

There is however, a third possibility – you may want strip out some annotations: not only ignore the annotation itself, but actually exclude the annotated content from the target. To do this you simply omit the instruction following the relevant operation test (that is, add a test with no corresponding action). To strip out all italic markup **and** content, for example, you could add the following line to the template:

```
{% elif op.name == "italic" %}
```

#### 4.3.1.2 CUE Print to Content Store

The template **`cue-print/cue-print-to-storyline/image`** converts a CUE Print image caption text to a pseudo-storyline version of a Content Store image story element.

The part of the template that is responsible for converting the caption looks like this:

```
{
    "name": "caption",
    "value": {{ caption.text_content|d("")|trim|tojson }},
    "annotations": []
}
```

The highlighted instruction in the **value** field extracts the text content (ignoring any markup), defaults to an empty string, trims any whitespace at the beginning and end of the text and then outputs the result as a JSON string.

In order to preserve any markup in the CUE Print text and allow corresponding storyline annotations to be created, the markup needs to be converted into pseudo-storyline operations. To achieve this, the **`value`** and **`annotations`** fields in the template need to be replaced with an **`ops`** field like this:

```
    "ops": [
{% with cutline = caption.content.data.ccitext.cutline.cutline_c.p|first %}
    {% for node in cutline recursive %}
        {% if node.is_text %}
            {
                "name": "",
                "text": {{ node.text_content|tojson }}
            }
        {% elif node.local_name == "bold" %}
            {
                "name": "bold",
                "sub": [
                    {{ loop(node) }}
                ]
            }
        {% endif %}
        {% if not loop.last %},{% endif %}
    {% endfor %}
{% endwith %}
    ]
```

The highlighted lines assign the content of the **first** paragraph in the CUE Print caption to a variable called **`cutline`** and pass it to the enclosed **`for`** loop. Any subsequent paragraphs (should they exist) are ignored. The long address of the caption paragraph reflects the deep XML structure used to represent captions in CUE Print:

```
<attribute group="ExtraInfo" kind="xml" name="CaptionTextAsXml">
  <content>
    <data format="text/xml">
       <cci:ccitext xmlns:cci="urn:schemas-ccieurope.com">
         ...
         <cci:cutline>
           <cci:cutline_c>
             <cci:p>
               Lorem ipsum <cci:bold>dolor sit amet</cci:bold>, consectetur adipiscing
               elit. Pellentesque quis lobortis ligula. Morbi hendrerit non purus sit
               amet volutpat. Mauris pulvinar velit ut augue vulputate, ac volutpat
               est molestie.
             </cci:p>
           </cci:cutline_c>
           ...
        </cci:cutline>
        ...
      </cci:ccitext>
    </data>
  </content>
</attribute>
```

The **`for`** loop recursively enumerates the contents of the paragraph:

Copyright © 2020-2023 Stibo DX A/S

```
{% for node in cutline recursive %}
    ...
{% endfor %}
```

The first **`if`** test selects every plain text node in the paragraph, and outputs the text as an operation with no name:

```
{% if node.is_text %}
    {
        "name": "",
        "text": {{ node.text_content|tojson }}
    }
```

The second test selects every **`<cci:bold>`** tag, wraps a bold **`operation`** its content, and recursively resubmits the content to the loop:

```
{% elif node.local_name == "bold" %}
    {
        "name": "bold",
        "sub": [
            {{ loop(node) }}
        ]
    }
```

The final **`else`** handles any other tags in the paragraph. It just recursively resubmits the content to the loop without wrapping an operation around it.

```
{% else %}
    {{ loop(node) }}
{% endif %}
```

The **`if`** at the end of the loop ensures that commas are inserted between the operations, as required by JSON syntax.

```
{% if not loop.last %},{% endif %}
```

As is the case with the Content Store – CUE Print template you can handle additional formats by adding more tests to the first **`if`** statement. For example:

```
{% elif node.local_name == "italic" %}
    {
        "name": "italic",
        "sub": [
            {{ loop(node) }}
        ]
    }
```

to convert **`italic`** tags to **`italic`** operations, or just:

```
{% elif node.local_name == "italic" %}
```

to strip out all **`italic`** content.

Copyright © 2020-2023 Stibo DX A/S

# 5 The Events Plugin

The Events plugin adds support to CUE Zipline for sending notification events to Amazon SQS or message brokers supporting the AMQP protocol, for example RabbitMQ.

The Events plug-in implements an event handler that should be configured as an external processor (or multiple, depending on the need), as described in section 3.10.5.

While external processors can be configured to poll resource URLs to monitor changes, the plug-in only supports CUE Content Store change logs as the event source and will ignore URL polling events.

The Events plug-in will generate a JSON object, with configurable properties, based on the information contained in updated CUE Content Store objects. The objects that will trigger notifications are limited only by the objects that are listed in the CUE Content Store change logs monitored by the configured processors.

For Amazon SQS, the plug-in will send the notification (JSON) object to a configured, pre-existing queue. The plug-in will not try to create the queue if it doesn't exist. This is intentional.

For AMQP protocol based (supporting) message brokers, the plug-in will send the notification object to a configured exchange, with an optional routing key. The plug-in will not try to create the exchange and/or perform any queue bindings. This is intentional.

## 5.1 Installation

Installing the Events plug-in for CUE Zipline doesn't require any prerequisites other than CUE Zipline and the Python libraries that are automatically added during installation.

The installation adds an example configuration file to the **`/etc/cue/zipline/processors.d`** folder, which must be edited as described in section 5.2.

After installing CUE Zipline, perform the command (with administrator rights) described for the target operating system below.

### 5.1.1 Ubuntu and Debian

To install the Events plug-in:

```
# apt-get install cue-zipline-events
```

### 5.1.2 RedHat and CentOS

To install the Events plug-in:

```
# yum install cue-zipline-events
```

Copyright © 2020-2023 Stibo DX A/S

## 5.2 Configuration

The example configuration file added to the **/etc/cue/zipline/processors.d** folder during installation contains two example processors, one for Amazon SQS and one for RabbitMQ (AMQP protocol based message brokers). The configurations are delimited by lines containing only "---". Just remove the configuration object you won't be using.

The general options in the configuration file, related to external processors, are described in section 3.10.5. Of the general options, just note that the **agent.factory** property must contain the value **cue.plugin.events:EventNotifier**.

The event plugin-specific configurations are described below.

### 5.2.1 Amazon SQS

For sending notifications to one or more Amazon SQS queues, the **configuration** configuration object should contain the following properties:

**type (required)**
The **type** property must contain the value "amazonsqs" to make the Events plug-in send notifications to an Amazon SQS queue.

**session (optional)**
The **session** configuration object defines properties that contain authorization credentials and AWS region information, needed to connect to the Amazon SQS service.

If some (or all) properties aren't included here, the Events plug-in will fall back to information configured for the AWS command-line tool, specifically in the **~/.aws/credentials** and **~/.aws/config** files.

The supported properties are:

**aws_access_key_id or aws_access_key_id_file (optional)**
Defines the ID of an access key of an (AMI) account, configured through the AWS console. The account must have write access to the queue(s) configured below.

When using **aws_access_key_id_file**, the value should be a file, from which the plug-in will read the first line as the access key ID.

**aws_secret_access_key or aws_secret_access_key_file (optional)**
Defines the secret key for the account. When using **aws_secret_access_key_file**, the value should be a file, from which the plug-in will read the first line as the secret access key.

**region_name (optional)**
Defines the AWS region to connect to (e.g., eu-central-1, us-west-1).

**profile_name (optional)**
If access key information is configured for the AWS command-line tool under a named profile (i.e., not the default profile), then use the **profile_name** property to identify the profile.

**targets (required)**
The **targets** property is a list of target queues, to which the plug-in will send events, when content is updated in any of the monitored CUE Content Store change logs.

Each entry in the list supports the following properties:

**`queue` (required)**

`queue` is a configuration object that contains properties needed to identify the target queue.

**`url` (optional)**

Uniquely identifies the queue, using the URL accessible from the AWS console.

**`queue_name` (optional)**

Identifies the queue by name. Used alone, this would target a queue with a name matching the value of this property in the account that created the access key used by the plug-in to access Amazon SQS.

Either `url` or `queue_name` must be specified.

**`queue_owner_aws_account_id` (optional)**

In case the queue exists in an account different from the one configured for the plug-in, then this property identifies that target account.

The access key used to connect to Amazon SQS must have rights to write events to the queue in this "foreign" account.

**`message_group_id` (optional)**

If the target queue is configured as a FIFO queue, then this property must define a message group ID.

**`message_body` (required)**

Defines a list of named properties that should be part of the event sent to the target queue.

Each entry in the list contains a `name` property that defines the name used for the property of the event.

The value of each property can be extracted from meta-data (e.g., `meta: id`), fields (`field: caption`), etc. The format of the entries is described in section 3.17.

The plug-in may export update events for multiple, very different object types (stories, sections, containers, etc.), so it is best to limit this to a common set of attributes. However, attributes that have no value for the updated object will just be left out of the message body.

### 5.2.2 RabbitMQ (AMQP Protocol)

The CUE Zipline Events plug-in can send notifications to message brokers supporting the AMQP protocol. Notifications are sent as JSON objects with properties that are configurable for each target.

The properties of the `configuration` object are as follows:

**`type` (required)**

To send notifications to an AMQP compatible message broker, this property must have the value "sqs".

**`connection` (required)**

The `connection` configuration object contains properties that define the connection to the message broker.

**`host` (required)**

Defines the host name of the server running the message broker.

**`port` (required)**

Defines the port number at which the message broker is listening.

**`use_tls` (optional)**

If the **`use_tls`** property is set to "true" or "yes", then the Events plug-in will use TLS when accessing the message broker.

**`credentials` (optional)**

The **`credentials`** configuration object should contain user credentials if authentication is needed to access the message broker.

Supported credentials properties are **`username`**, **`password`**, and **`password_file`**.

**`targets`**

The **`targets`** property is a list of target queues, to which the plug-in will send events, when content is updated in any of the monitored CUE Content Store change logs.

Each entry in the list supports the following properties:

**`exchange` (required)**

The name of the exchange to send events to.

**`routing_key` (required)**

The routing key for events sent to this exchange.

**`message_body` (optional)**

Defines a list of named properties that should be part of the event sent to the target queue.

Each entry in the list contains a **`name`** property that defines the name used for the property of the event.

The value of each property can be extracted from meta-data (e.g., **`meta: id`**), fields (**`field: caption`**), etc. The format of the entries is described in section 3.17.

The plug-in may export update events for multiple, very different object types (stories, sections, containers, etc.), so it is best to limit this to a common set of attributes. However, attributes that have no value for the updated object will just be left out of the message body.

Copyright © 2020-2023 Stibo DX A/S

# 6 The Zip Export Plugin

The Zip Export plugin allows CUE Zipline to support export of content from the Content Store using the Newsgate Plugin. The Newsgate plugin can be used to export a content item as a zip file containing the content item itself (in the Content Store's XML-based syndication format) together with all the additional resources it references (media files and so on).

This plugin provides an event handler for CUE Zipline that makes use of the Newsgate Plugin's web service to export content items and write them either to disk or to an AWS S3 bucket. You can use it, therefore, to set up an external processor that will listen for updates in all or part of a publication, and export a zip file every time a content item is modified.

The zip files created by the Zip Export plugin are timestamped so that subsequent exports of the same content item will not overwrite previous exports, but result in the creation of new files. It is the consumer's responsibility to remove old versions of export files when they are no longer needed.

The following tasks must be carried out to make use of the Zip Export plugin:

1. Install the Zip Export plugin, as described in section 6.1.
2. Create one or more destinations for the exported zip files. (The target disk folder or AWS bucket must exist, it will not be created by the Zip Export plugin.)
3. Configure one or more external processors (one for each export destination) as described in section 6.2.

## 6.1 Installation

Before installing the Zip Export plugin, ensure that:

- CUE Zipline itself is correctly installed
- The Newsgate plugin is correctly installed, and the **newsgate-webservice** has been deployed

Once you have done that, the installation procedure is:

1. Log in as **root**, and enter the appropriate installation command:

   **For Ubuntu or Debian:**

   ```
   # apt-get install cue-zipline-zip-export
   ```

   **For Redhat or CentOS:**

   ```
   # yum install cue-zipline-zip-export
   ```

2. Configure the plugin, as described in section 6.2.
3. Restart CUE Zipline.

## 6.2 Configuration

During installation an example configuration file called **zip-export.yaml.example** is copied to the **/etc/cue/zipline/processors.d/** folder. Any **.yaml** files in this folder are automatically

Copyright © 2020-2023 Stibo DX A/S

detected by CUE Zipline and used as processor configurations, as described in section 3.18. So to configure the Zip Export plugin you need to:

1. Copy or rename **`/etc/cue/zipline/processors.d/zip-export.yaml.example`** to **`/etc/cue/zipline/processors.d/zip-export.yaml`**.
2. Edit **`/etc/cue/zipline/processors.d/zip-export.yaml`** to meet your requirements.

If you want to export to multiple destinations, then you will need to repeat this process, using different names for the resulting **`.yaml`** files.

The example file contains an external processor configuration, as described in section 3.10.5. Below is a description of how to configure the Zip Export-specific parts of the file:

**`agent/factory`**
Must be set to **`cue.plugin.zip_export:ZipExporter`**.

**`agent/configuration`**
The content of the **`configuration`** object depends on whether you want to export to disk or to an S3 bucket, as described in the following sections.

### 6.2.1 Export to Disk

For export to a folder on the local disk, the **`configuration`** object must contain the following properties:

**`type` (required)**
Must be set to **`disk`**.

**`dir` (required)**
Path to the destination folder. This folder must exist, it will not be created by the plugin.

**`newsgate_webservice_endpoint/url` (required)**
The URL of the Newsgate web service's endpoint — most probably something like **`https://`***content-store-host***`/newsgate-webservice`**.

**`newsgate_webservice_endpoint/max_chunk_size` (optional)**
The plugin downloads data from the Newsgate web service in chunks, in order to limit memory usage in the case of very large zip files. This property sets the maximum chunk size in bytes. The default value is **`5242880`** (5mb).

### 6.2.2 Export to S3

For export to an Amazon S3 Bucket, the **`configuration`** object must contain the following properties:

**`type` (required)**
Must be set to **`s3`**.

**`bucket_name` (required)**
The name of the destination S3 bucket. This bucket must exist, it will not be created by the plugin.

**`key_prefix` (optional)**
A key prefix for all the zip files uploaded to Amazon S3. Key prefixes are effectively S3 bucket "folders" and look like folder paths. The default value is empty, which means that exported files will be uploaded to the bucket's root folder.

**`newsgate_webservice_endpoint` (required)**
The URL of the Newsgate web service's endpoint — most probably something like **`https://`***content-store-host***`/newsgate-webservice`**.

**`session` (optional)**
The **`session`** configuration object contains the authorization credentials and AWS region information needed to connect to the S3 service. It is the same configuration object as is used for configuring Amazon SQS in the Events plugin — see section 5.2.1 for details.

Copyright © 2020-2023 Stibo DX A/S 

# 7 The Sophi Plugin

Sophi is a third-party web service (https://www.sophi.io/) that uses AI to offer automated content curation based on a sophisticated analysis of both website content and usage data. It can be used with CUE to provide automated desking of content on section pages and realtime feedback regarding the performance of published content. The Sophi plugin is a CUE Zipline plugin that provides the "plumbing" needed to support automated desking in CUE. It consists of two processors:

**Sophi Content Feed**
: This processor monitors the Content Store for publishing events. Every time a new content item is published or a published content item is modified, the content is sent to Sophi for analysis.

**Sophi List Updater**
: This processor polls Sophi at regular intervals (say every 10 minutes) for curation recommendations and updated performance data. It's called a "list updater" because CUE lists are used to hold the content items that Sophi recommends for desking. The processor updates the relevant lists in the Content Store and triggers republishing of the section pages on which the lists are desked.

For more general information about Sophi, visit the Sophi website. For more information about the Sophi-based features in CUE and how to enable them, see Automated Curation With Sophi.

## 7.1 Installation

Before installing the Sophi plugin, ensure that:

- CUE Zipline itself is correctly installed
- You have set up a Sophi account, and have the credentials CUE Zipline will need to access the Sophi web service

Once you have done that, the installation procedure is:

1. Log in as **`root`**, and enter the appropriate installation command:

   **For Ubuntu or Debian:**

   ```
   # apt-get install cue-zipline-sophi
   ```

   **For Redhat or CentOS:**

   ```
   # yum install cue-zipline-sophi
   ```

2. Configure the plugin, as described in section 7.2.
3. Restart CUE Zipline.

## 7.2 Configuration

During installation an example configuration file called **`sophi-processor.yaml.example`** is copied to the **`/etc/cue/zipline/processors.d/`** folder. Any **`.yaml`** files in this folder are automatically detected by CUE Zipline and used as processor configurations, as described in section 3.18. So to configure the Sophi plugin you need to:

1. Copy or rename **/etc/cue/zipline/processors.d/sophi-processor.yaml.example** to **/etc/cue/zipline/processors.d/sophi-processor.yaml**.
2. Edit **/etc/cue/zipline/processors.d/sophi-processor.yaml** to meet your requirements.

The Sophi configuration file contains the configurations of two **external processors**. For general information about external processor configurations, see section 3.10.5. Each processor configuration is preceded by a YAML object delimiter:

```
---
```

### 7.2.1 Sophi Content Feed Configuration

The Sophi Content Feed processor monitors the Content Store for publishing events, and passes all published and republished content items to Sophi for analysis.

The Sophi Content Feed configuration has the following overall structure:

```
name:
source:
  urls:
  username:
  password-file:
filter:
agent:
  factory:
  configuration:
    endpoint:
    environment:
    username:
    password-file:
    client-id:
    app-id-template:
```

Some of the properties in the supplied example file are predefined and do not need to be changed. Edit the configuration as follows

**name**
: The name of the processor. It appears in log messages, but is not used for anything else.

**source/urls**
: An array of Content Store change log URLs, one for each publication that is to be controlled by Sophi. For example:

```
source:
  urls:
    - https://content-store-host/webservice/escenic/changelog/publication/
tomorrow-online
    - https://content-store-host/webservice/escenic/changelog/publication/
living-online
```

**source/username**
: A Content Store user name. This user name will be used to log into the Content Store web service and access the change log. For example:

```
source:
  username: sophi_content_store
```

Read access only (a **reader** role) is sufficient for this user.

**`source/password_file`**

The path of a file containing the Content Store password. For example:

```
source:
  password_file: /var/run/secrets/sophi_content_store_password
```

You can, if you wish replace this property with a **`password`** property containing the actual password:

```
source:
  password: notverysecret
```

This is, however, not recommended. It is better to keep your passwords in separate files that are not committed to your code repository, for security reasons.

**`filter`**

The stream of events read from the change logs monitored by the processor is by default filtered as defined in the main CUE Zipline configuration file. You can, however, also include a filter specification in this configuration file. Any filter specified here will be applied in addition to the main filter. The supplied example configuration contains the following processor-specific filter:

```
filter:
  - story_type:
      - storyline
```

This limits the processor to only handling storylines. The Sophi plugin only supports storyline content.

See section 3.9 for general information about how to define filters.

**`agent/factory`**

The identifier of the code module responsible for passing events and content to Sophi. The predefined setting of **`cue.plugin.sophi:SophiContentFeed`** must not be modified:

```
agent
  factory: cue.plugin.sophi:SophiContentFeed
```

**`agent/configuration/endpoint`**

The URL of the Sophi endpoint to which events and content are sent. This endpoint is specified by Sophi:

```
agent:
  configuration:
    endpoint: https://collector.sophi.io
```

**`agent/configuration/environment`**

One of the following three keywords, identifying the type of installation you are configuring the plugin for:

- **`dev`** (development)
- **`stg`** (staging)
- **`prod`** (production)

If you are configuring a development system, for example, then you should enter:

```
agent:
  configuration:
    environment: dev
```

Copyright © 2020-2023 Stibo DX A/S

**agent/configuration/username**

The agent needs Content Store credentials in order to be able to retrieve additional information about updated content items. This property is usually set to the same value as **source/username** (above), but you can use a different account if you wish.

**agent/configuration/password_file**

The password for the user specified in **agent/configuration/password_file**. Here too, you can replace **password_file** with **password**, although it is not recommended to do so.

**agent/configuration/client-id**

A string provided by Sophi that identifies your Sophi account:

```
agent:
  configuration:This
    client-id: sophi-client-id
```

**agent/configuration/app-id-template**

Sophi uses **application IDs** to identify all the uploaded events and content belonging to specific publications. This property specifies a template that generates unique **application IDs** for all your publications. The required format and structure of application IDs is specified by Sophi.

```
agent:
  configuration:
    app-id-template: "{client_id}:{publication_name}_{hostname}:cms"
```

You should not need to modify the supplied template.

In case of special requirements, you can replace the **app-id-template** property with an **app-ids** array as follows:

```
agent:
  configuration:
    app-ids:
      news-publication: "sophi_client_id:tomorrow-online_content-store-host:cms"
      sport-publication: "sophi_client_id:living-online_content-store-host:cms"
```

You can use this alternative method if for some reason it is not possible to automatically generate the required IDs using a template.

### 7.2.2 Sophi List Updater Configuration

The Sophi List Updater processor polls Sophi at regular intervals for curation recommendations. It then updates the relevant publication sections in the Content Store and republishes them. Each publication section that is to be wholly or partly controlled by Sophi must have one or more lists intended to hold Sophi recommendations. Those lists must be desked as required on the section's active section page. The Sophi List Updater will then fill those lists with recommendations it retrieves from Sophi

The Sophi List Updater configuration has the following overall structure:

```
name:
source:
  polling:
    urls:
    auth:
      oauth:
        token_url:
        client_id:
```

Copyright © 2020-2023 Stibo DX A/S

```
        client_secret_file:
        audience:
    interval:
agent:
  factory: cue.plugin.sophi:SophiListUpdater
  configuration:
    content_store:
      url:
      username:
      password_file:
    default_list:
    publications:
      domain_name:
```

Some of the properties in the supplied example file are predefined and do not need to be changed. Edit the configuration as follows

**name**
The name of the processor. It appears in log messages, but is not used for anything else.

**source/polling/urls**
This property must contain an array of URLs, one for each list controlled by Sophi:

```
source:
  polling:
    urls:
      - https://site-automation-api.ml.sophi.works/curatedHosts/tomorrow-
online.com/curator?page=ece_frontpage&widget=Automated+List+1
      - https://site-automation-api.ml.sophi.works/curatedHosts/tomorrow-
online.com/curator?page=news&widget=Automated+List+1
      - https://site-automation-api.ml.sophi.works/curatedHosts/living-
online.com/curator?page=ece_frontpage&widget=Automated+List+1
```

Currently, the URLs must have the following form, as shown in the example above:

```
https://site-automation-api.ml.sophi.works/curatedHosts/domain-name/curator?
page=section-name&widget=list-name
```

where:

- *domain-name* is the domain name of the publication to be updated.
- *section-name* is the unique name of section owning the list to be updated.
- list-name is the name of the list to be updated.

> The URLs shown above are based on Sophi's requirements at the time of writing. Stibo DX cannot guarantee that they will continue to be correct.

**source/polling/auth/oauth/token_url**
The URL of the Sophi endpoint to which OAuth credentials are sent, in order to obtain an authentication token:

```
source:
  polling:
    auth:
      oauth:
        token_url: sophi-token-endpoint-url
```

This URL must be supplied by Sophi.

**source/polling/auth/oauth/client_id**

An OAuth client ID (not the same as the **client-id** property used in the Content Feed configuration).

```
source:
  polling:
    auth:
      oauth:
        client_id: oauth-client-id
```

This ID must also be supplied by Sophi.

**source/polling/auth/oauth/client_secret_file**

The path of a file containing an OAuth secret. For example:

```
source:
  polling:
    auth:
      oauth:
        client_secret_file: /etc/cue/zipline/secrets/sophi-client-secret
```

The OAuth secret in the file must be supplied by Sophi. You can, if you wish replace this property with a **client_secret** property containing the actual secret:

```
source:
  polling:
    auth:
      oauth:
        audience: notverysecret
```

This is, however, not recommended. It is better to keep secrets in separate files that are not committed to your code repository, for security reasons.

**source/polling/auth/oauth/audience**

The URL of the site that the supplied OAuth credentials are intended to give access to:

```
source:
  polling:
    auth:
      oauth:
        audience: https://api.sophi.works
```

**source/polling/interval**

Specifies how frequently the List Updater is to request updates from Sophi, for example:

```
source:
  polling:
    interval:
      minutes: 10
```

A request is sent to every URL specified in **source/polling/urls** once in each time interval, and if you have specified more than one URL, then the requests are spread evenly across the interval. If, for example, you specify an interval of 10 minutes as shown, and specify 10 URLs in **source/polling/urls**, then one request will be sent each minute.

You can replace the **minutes** property with a different time unit - **weeks**, **days**, **hours**, **minutes**, or **seconds**. For example:

```
source:
  polling:
    interval:
      hours: 1
```

Copyright © 2020-2023 Stibo DX A/S 

**`agent/factory`**

The identifier of the code module responsible for polling Sophi and updating the Content Store. The predefined setting of **`cue.plugin.sophi:SophiListUpdater`** must not be modified:

```
agent
  factory: cue.plugin.sophi:SophiListUpdater
```

**`agent/configuration/content_store/url`**

The URL of the Content Store webservice. For example:

```
agent:
  configuration:
    content_store:
      url: https://content-store-host/webservice/
```

**`agent/configuration/content_store/username`**

A Content Store user name. This user name will be used to log into the Content Store web service and update lists. For example:

```
agent:
  configuration:
    content_store:
      username: sophi_content_store
```

Read/write access (an **editor** role) is required for this user.

**`source/password_file`**

The path of a file containing the Content Store password. For example:

```
agent:
  configuration:
    content_store:
      source:
        password_file: /var/run/secrets/sophi_content_store_password
```

You can, if you wish replace this property with a **`password`** property containing the actual password:

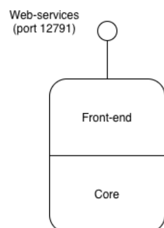
```
agent:
  configuration:
    content_store:
      source:
        password: notverysecret
```

This is, however, not recommended. It is better to keep your passwords in separate files that are not committed to your code repository, for security reasons.

# 8 Clustering

CUE Zipline is composed of two parts:

- The front end, which provides web services to external clients such as the CUE editor, drop resolvers, CUE Print and so on.
- The core, which monitors the Content Store, waiting for changes to content items that have "shadow" content in external systems (CUE Print and DC-X, plus other systems via NewsML export).




When running a CUE Zipline cluster, the front end web services should be load balanced, in order to distribute incoming client requests between the instances in the cluster.

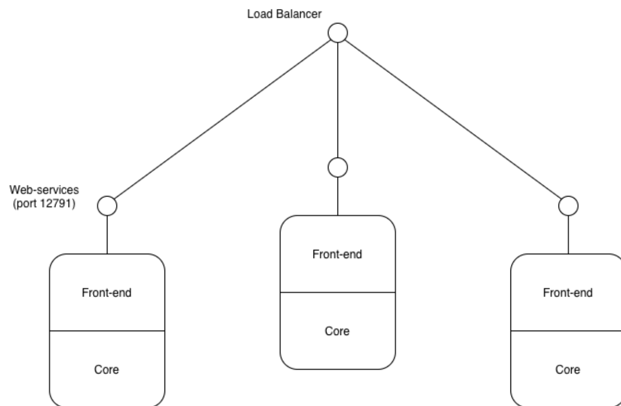
The core activity of monitoring the Content Store for changes cannot, however, be load balanced in this way. Only one CUE Zipline instance (called the "active" instance) monitors the Content Store. The other instances in the cluster are said to be "inactive", even though they are available for processing front end web service requests.

## 8.1 Front End

All external systems that send requests to CUE Zipline send them to its web service endpoint. Any request sent to a particular CUE Zipline instance's endpoint will be processed on that instance. In order to distribute the load between multiple CUE Zipline instances, therefore, the requests must be actually directed to those specific instances. The easiest way to do this is to enable load balancing in

Copyright © 2020-2023 Stibo DX A/S

the reverse proxy that you should already be using to handle TLS termination for CUE Zipline (see section 2.4).




With the reverse proxy handling load balancing, it is then only necessary to specify the proxy as the CUE Zipline endpoint when configuring external client systems (that is, CUE Print, drop resolvers and the CUE editor).

### 8.1.1 Load Balancing

Web service requests to the front end can be load balanced, using any standard reverse proxy. The example provided in section 2.4 shows a proxy configuration for a single instance CUE Zipline installation.

For **nginx**, the only required change is to create an **upstream** definition for the CUE Zipline instances and then reference those back ends in the **proxy_pass** statement. For example:

```
upstream backends {
    server zipline01:12791;
    server zipline02:12791;
  }

  server {
    ...
    location ~ ^/cue-print-zipline/(index.xml|escenic/text|escenic/convert/default) {
      ...
      proxy_pass http://backends;
      ...
    }
  }
```

The **upstream** configuration should list the web service addresses for each instance in the CUE Zipline cluster. This address is the one configured in the section 3.4 configuration object. The default is port 12791 on the server the instance is running on.

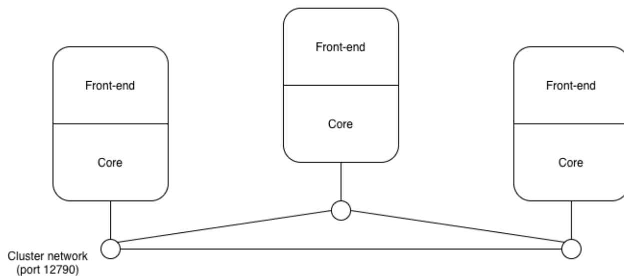
This default configuration will spread the load between your CUE Zipline instances using a round-robin algorithm. **nginx** does however, offer alternative load distribution algorithms.

The protocol scheme specified in the **proxy_pass** declaration must be **http** as shown above, since CUE Zipline only supports HTTP.

Copyright © 2020-2023 Stibo DX A/S

## 8.2 Core

The CUE Zipline cores communicate with each other via private network connections, using the network addresses defined in the **members** list configured in the section 3.14 configuration object.




The **members** list is a list of host name and port pairs for each instance in the cluster. Each instance opens connections to the other instances by connecting to these network addresses.

The network address of each instance is defined by the **listen_address** property of the **cluster** configuration object. The listen address may be specified as a wild-card (e.g. **0.0.0.0**) in order to listen on all network interfaces.

The port number of the cluster **listen_address** **must** be different from the **server** port number, because the communication protocol is different. By default, the server port is **12791** and the cluster listen address port is **12790**.

The host names and addresses specified in the **members** list may not be specified as wild cards. They **must** be host names that can be resolved or IP addresses, and they must be accessible to the other instances in the cluster.

### 8.2.1 Negotiations

On startup, or whenever an active instance shuts down, the cluster instances negotiate to determine which one will be active.

Priority is given to the instance with the highest ID (as specified in the **instance_id** property in the **cluster** configuration object). If not all instances are running during the negotiation process, however, then another instance may be designated active instead.

If **instance_id**s are not explicitly set, then instances are assigned random IDs each time they are started. If instances are regularly re-started, this introduces some randomness with regard to which instance is given priority during negotiation.

After negotiation, all the non-active instances will enter the inactive state, in which they do not process any Content Store updates. If the active instance shuts down or disappears, the idle instances will re-enter negotiations to find a new active instance.

### 8.2.2 Change Log Monitoring

As updates are processed in the active instance, its progress is automatically shared with the inactive instances in the cluster. The inactive instances store this progress data locally, in order to be able to continue processing should they become the active instance.

Copyright © 2020-2023 Stibo DX A/S 

Processing of updates detected in the change log are only performed on the active instance. No processing is performed on the inactive instances.

### 8.2.3 Example

The following example is based on a cluster of three instances, each running on its own server. The **`instance_id`** property is different on each instance, but they all use the same **`listen_address`** (the **`0.0.0.0`** wild-card) and the **`members`** list **must** be the same on each server.

Here, for example, is the configuration for the first server:

**zipline_01**

```
cluster:
  instance_id: zipline_01
  listen_address: 0.0.0.0:12790
  members:
    - ziplinesrv01.cust.cue.cloud:12790
    - ziplinesrv02.cust.cue.cloud:12790
    - ziplinesrv03.cust.cue.cloud:12790
```

A string comparison is used when comparing instance IDs during active instance negotiation, so **`zipline_9`** would have higher priority than **`zipline_10`**. That is why zero-padded number are used to construct the example IDs.

The listen address is set to port **`12790`** on all network interfaces of the server running this instance. Since this is the default value it could be omitted.

The member list includes the domain name and port number of each cluster instance, including **this** instance. This local instance could be omitted, but including it causes no problems and makes maintaining the configuration files easier (they can all be identical apart from the **`instance_id`** setting).

The other two configuration files, then, look like this:

**zipline_02**

```
cluster:
  instance_id: zipline_02
  listen_address: 0.0.0.0:12790
  members:
  - ziplinesrv01.cust.cue.cloud:12790
  - ziplinesrv02.cust.cue.cloud:12790
  - ziplinesrv03.cust.cue.cloud:12790
```

**zipline_03**

```
cluster:
  instance_id: zipline_03
  listen_address: 0.0.0.0:12790
  members:
    - ziplinesrv01.cust.cue.cloud:12790
    - ziplinesrv02.cust.cue.cloud:12790
    - ziplinesrv03.cust.cue.cloud:12790
```

As mentioned in section 3.14, you can define a **`ZL_CLUSTER_LISTEN_ADDRESS`** environment variable to hold the instance ID on each server. Since the default value of the **`listen_address`** property is **`0.0.0.0`**, this means you can in fact use the same configuration file on all instances:

Copyright © 2020-2023 Stibo DX A/S

```
cluster:
  members:
  - ziplinesrv01.cust.cue.cloud:12790
  - ziplinesrv02.cust.cue.cloud:12790
  - ziplinesrv03.cust.cue.cloud:12790
```

The **zipline_01** instance could then be started as follows, for example:

```
$ export ZL_CLUSTER_LISTEN_ADDRESS=zipline_01

$ zipline
```

# 9 Logging

CUE Zipline generates log output using the standard Python logging subsystem. The log output is designed to provide information for troubleshooting integration issues.

Four different levels of log message are generated by CUE Zipline:

| Level | Description |
|---|---|
| **ERROR** | Generated for events that are considered to be processing errors, such as when a back-end service responds with an error to a request that is expected to succeed. |
| **WARNING** | Generated for less serious adverse events, such as when a back-end service responds with an error to a request that is not expected to always succeed. |

Copyright © 2020-2023 Stibo DX A/S

| Level | Description |
|---|---|
| **`INFO`** | Generated for normal events, describing what CUE Zipline is doing. When handling a content item update, for example, **`INFO`** messages will be logged when the event is received, when the items required to process the request are resolved, when back-end requests are sent and so on. |
| **`DEBUG`** | More detailed messages describing how CUE Zipline handles events. |

Copyright © 2020-2023 Stibo DX A/S

| Level | Description | |
|---|---|---|
| | **DEBUG** messages may be logged, for example, for each of the individual requests sent to a back-end service while resolving a content item for processing. | |

You can control which of these messages are actually written to file, where they are sent, how long they are kept and so on by configuring the logging system.

## 9.1 Configuration

CUE Zipline is delivered with a default logging configuration which you will find in the **logging** section of the configuration file (**/etc/cue/zipline/zipline.yaml**):

```
logging:
  version: 1
  formatters:
    precise:
      format: '%(asctime)s - %(levelname)-5s - %(name)s - %(message)s'
      style: '%'
  handlers:
    file:
      class: logging.handlers.TimedRotatingFileHandler
      formatter: precise
      filename: /var/log/zipline/zipline.log
      when: midnight
      level: DEBUG
      encoding: UTF-8
  root:
    level: DEBUG
    handlers:
      - file
  loggers: {}
```

This simple configuration writes **all** messages to the file **/var/log/zipline/zipline.log**. The file is overwritten at midnight each day.

Copyright © 2020-2023 Stibo DX A/S

In the installation **contrib** folder (**/usr/share/cue/cue-zipline/contrib/**), you will find a more sophisticated logging configuration in a file called **logging-config.yaml**:

```
version: 1

formatters:
  precise:
    format: '%(asctime)s - %(levelname)-5s - %(name)s - %(message)s'
    style: '%'

handlers:
  debugfile:
    class: logging.handlers.TimedRotatingFileHandler
    formatter: precise
    filename: /var/log/zipline/zipline.debug.log
    backupCount: 7
    when: midnight
    level: DEBUG
    encoding: UTF-8
  file:
    class: logging.handlers.TimedRotatingFileHandler
    formatter: precise
    filename: /var/log/zipline/zipline.log
    backupCount: 7
    when: midnight
    level: ERROR
    encoding: UTF-8

# Root logger configuration
root:
  level: DEBUG
  handlers:
    - debugfile
    - file

loggers:
  chardet.charsetprober:
    level: ERROR
  cue.zipline.text.transform_text.TextTransformer:
    level: INFO
  cue.concurrent.actor.Actor:
    level: INFO
  cue.zipline.audit:
    level: CRITICAL
```

This configuration only writes **ERROR** messages to **/var/log/zipline/zipline.log**, but in addition writes all messages to **/var/log/zipline/zipline.debug.log**. In addition, the **backupCount** settings of **7** means that 7 backup copies of each log file are retained, so that you always have all messages from the preceding week available.

On startup, CUE Zipline looks in two places for a logging configuration:

- First, it looks for a standalone logging configuration in **/etc/cue/zipline/logging-config.yaml**. If it finds a configuration here, then that is the one it uses.
- If **/etc/cue/zipline/logging-config.yaml** does not exist, then it looks for a **logging** section in **/etc/cue/zipline/zipline.yaml** and uses the configuration it finds there.

Copyright © 2020-2023 Stibo DX A/S

- If it cannot find a logging configuration in either place, then CUE Zipline uses its internal defaults, which provide minimal functionality.

You can therefore choose where to keep your logging configuration. Either edit the default configuration in **`/etc/cue/zipline/zipline.yaml`** to meet your requirements, or copy **`/usr/share/cue/cue-zipline/contrib/logging-config.yaml`** into the **`/etc/cue/zipline/`** folder and edit that instead. If you decide to use a standalone **`logging-config.yaml`** file, then it is a good idea to remove the logging section from **`/etc/cue/zipline/zipline.yaml`** to avoid confusion.

For detailed information about the logging configuration format, see here.

Copyright © 2020-2023 Stibo DX A/S

# 10 Monitoring

CUE Zipline incorporates a monitoring service that supplies reports about CUE Zipline's activities. Two kinds of report are generated:

- On-demand JSON reports returned in response to requests sent to CUE Zipline's monitoring endpoint.
- Automated plain text reports generated at specified intervals and written to the CUE Zipline log file.

For information on configuring automated reports, see section 3.15.

For information on how to request JSON reports, see section 10.1, and for information on the structure of the JSON reports, see section 10.2.

## 10.1 Requesting a Report

To retrieve a report from CUE Zipline:

1. Send an initial **GET** request to CUE Zipline's root endpoint:

   ```
   curl http://localhost:12791/cue-print-zipline/
   ```

   CUE Zipline will respond by returning an XML discovery document containing the URLs of its various endpoints, including the the monitoring endpoint. The monitoring endpoint can be identified by the relation type **monitoring**:

   ```
   <endpoint xmlns="http://xmlns.ccieurope.com/ngece-bridge"
         xml:base="http://localhost:12791/cue-print-zipline/" ...>
       ...
       <link rel="monitoring" href="monitoring"/>
   </endpoint>
   ```

2. Retrieve the monitoring endpoint URL fragment from the **link** element's **href** attribute and construct a new URL (**http://localhost:12791/cue-print-zipline/monitoring** in this case):
3. Submit a new request to the monitoring endpoint:

   ```
   curl http://localhost:12791/cue-print-zipline/monitoring
   ```

   CUE Zipline will then return a report on its recent activities.

By default, the report will contain information about CUE Zipline's activity during the previous 15 minutes. You can, however modify the reporting period in two ways:

- You can set a different default reporting period using the **default_range** configuration parameter (see section 3.15).
- You can specify the required reporting period for an individual request by appending a **period** parameter to it:

  ```
  curl http://localhost:12791/cue-print-zipline/monitoring?period=24h
  ```

Copyright © 2020-2023 Stibo DX A/S

You can specify the reporting period in either hours (**24h**), minutes (**30m**) or seconds (**30s**). The period always represents the time immediately before the request was submitted: the last 24 hours, 30 minutes or 30 seconds. You can, if you wish request reporting for multiple periods in a single request:

```
curl http://localhost:12791/cue-print-zipline/monitoring?period=24h,30m,30s
```

The specified periods must be separated by commas.

## 10.2 Report Structure

The monitoring report is a JSON document with the following overall structure:

```
{
  "items": [],
  "query": {
    "period": ["24h","30m","30s"]
  }
}
```

The report has two top-level properties:

**items**
: An array containing the main body of the report.

**query**
: Contains the parameters from the query that initiated the report. This information may be useful in the development of monitoring plug-ins.

The **items** array contains a list of entries, each containing information about some aspect of CUE Zipline operation during the requested period(s):

```
{
  "component": "...",
  "data": ...
}
```

Each entry has two properties:

**component**
: The name of the CUE Zipline report section (often, but not always, the name of a component of the CUE Zipline system.

**data**
: Information about this component. The structure of this property depends on the type of component. In most cases, the property is an array that can contain one report object for each requested time period. In some cases, however, (currently only for the **state** component) an array is not required, and **data** is a single object.

## 10.3 Components

A report can contain the following components:

Copyright © 2020-2023 Stibo DX A/S 

### 10.3.1 Restarts

This component contains information about restarts during the requested period(s).

```
{
  "component": "/restarts",
  "data": [
    {
      "range": "15m",
      "kills": 0,
      "restarts": 1
    }
  ]
}
```

**range**
The time period for which the data is aggregated.

**restarts**
The number of times this instance of CUE Zipline has been restarted in the specified period.

**kills**
The number of the reported **restarts** that were forced.

### 10.3.2 State

The current state (active/inactive) of this CUE Zipline instance.

```
{
  "component": "/state",
  "data": {
    "state": "active"
    "remotes": [
      "01_zipline"
    ],
    "disconnected": [],
  }
}
```

**state**
The state of this CUE Zipline instance. **active** means it is actively processing updates from Content Store, **inactive** means it is not.

In a CUE Zipline cluster only one instance is active at any given time: the other instances will report that they are inactive. If the active instance is shut down, one of the inactive instances will change state to "**active** and start processing updates.

**remotes**
The IDs of all connected remote instances in a CUE Zipline cluster. In a cluster with three members, this list should always contain two entries (on all three instances). If this is not the case, then one or more instances are not communicating properly.

**disconnected**
The IDs of all instances that have been connected in the past, but are not currently connected. If this list contains entries and all your CUE Zipline instances are running, then the instances are not communicating properly.

Copyright © 2020-2023 Stibo DX A/S

### 10.3.3 Processors

A component is generated for each processor running on the CUE Zipline instance:

- **`/processors/cue_print`** for CUE Print integration
- **`/processors/dcx`** for DC-X integration
- **`/processors/newsml`** for NewsML export

All these components have the same structure:


```
{
  "component": "/processors/newsml",
  "data": [
    {
      "range": "15m",
      "count": 6,
      "updates": {
        "success": 6
      },
      "waiting_time": {
        "average": 2.7780989011128745,
        "max": 6.311340093612671,
        "min": 0.5846478939056396
      }
      "processing_time": {
        "average": 0.7568506002426147,
        "max": 1.651845932006836,
        "min": 0.04902076721191406
      },
    }
  ]
}
```


**`range`**
: The length of the time period.

**`count`**
: The number of events processed in the time period. This does not include events filtered out either by the global filter or a processor-specific filter.

**`updates`**
: The outcomes of the processed events. The possible outcomes are:

  - **`success`** (update completed, successful)
  - **`failure`** (update failed during processing)
  - **`partial`** (update completed, partly successful)
  - **`pending`** (update still in progress)

**`waiting_time`**
: The length of time elapsed before events were processed during this time period in seconds. Three different values are reported: average, minimum and maximum waiting times.

  The waiting time includes the common processing done by CUE Zipline to resolve an updated content item, so it can never be zero.

Copyright © 2020-2023 Stibo DX A/S

**`processing_time`**

The length of time spent processing each event, in seconds. Three different values are reported: average, minimum and maximum processing times.

# 11 Recording Catch Files

CUE Zipline provides two tools for controlling the recording and saving of CUE Print **catch files**. Catch files are diagnostics files that can be recorded and saved during CUE Print sessions. In general, catch files are recorded by starting a new CUE Print session with catch file recording enabled. The tools provided with CUE Zipline are:

- A command line utility called **`zl-catch`**
- A web API

## 11.1 zl-catch

**`zl-catch`** provides a convenient way of recording catch files for analyzing communication between CUE Print and CUE Zipline, from the CUE Zipline host. When you enable/disable recording with **`zl-catch`**, **`zl-catch`** restarts the CUE Print session for you with the requested catch file setting.

The syntax of the **`zl-catch`** command is:

```
zl-catch [-h] [-f configuration-file-path] [{status,enable,disable,save}]
```

**Options**

**`-h`**
Displays help

**`-f`** ***configuration-file-path***
The path of the CUE Zipline configuration file. **`zl-catch`** needs access to the configuration file in order to retrieve the URL of the CUE Print endpoint. If the CUE Zipline configuration file is stored in a standard location then this option is not required.

**Subcommands**

**`status`**
Displays the current status of catch file recording for CUE Zipline. This is the default action.

**`enable`**
Starts a new CUE Print session with catch recording enabled. If recording was already enabled, a new session is still started and any activity that was already recorded is discarded.

**`disable`**
Starts a new CUE Print session with catch recording disabled.

**`save`**
Instructs CUE Print to save a catch file containing any activity recorded since the last session restart. The name of the file is written to the console. It is also written to the CUE Zipline log file as an **`INFO`** level message. The catch file is of course saved on the CUE Print host, not the CUE Zipline host.

Copyright © 2020-2023 Stibo DX A/S 

## 11.2 The Catch File API

CUE Zipline exposes a catch file API at *cue-zipline*/**`cue-print/catch`**. This API is used by the **`zl-catch`** command, but is also available for use by remote management applications.

The response from the API endpoint is a JSON object containing the current recording status and links that can be used to perform actions appropriate for the current state. For example: executing

```
curl http://localhost:12791/cue-print-zipline/cue-print/catch
```

will return the following if catch file recording is currently enabled:


```
{
    "self": "http://localhost:12791/cue-print-zipline/cue-print/catch",
    "home": {
        "data": {"catch": "enabled", "status": "ok"},
        "actions": [
            {
                "name": "home",
                "href": "http://localhost:12791/cue-print-zipline/cue-print/catch",
                "rel": ["home", "status"],
                "method": "GET"
            },
            {
                "name": "disable",
                "href": "http://localhost:12791/cue-print-zipline/cue-print/catch/
disable",
                "rel": ["disable"],
                "method": "PUT"
            },
            {
                "name": "save",
                "href": "http://localhost:12791/cue-print-zipline/cue-print/catch/
save",
                "rel": ["save"],
                "method": "PUT"
            }
        ]
    }
}
```


Note that in this case, no **`enable`** action is offered, since recording is already enabled.

Copyright © 2020-2023 Stibo DX A/S 