CUE Zipline

User Guide

1.10.0-6

Table of Contents

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

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

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.

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.

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.

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

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

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.

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 and CentOS

To install CUE Zipline:

1. Install the CUE Zipline dependencies:

```
# yum install -y \
  findutils \
  gcc \
  python3 \
  python3-devel \
  python3-pip
```

2. On RedHat 7 only (this step is not required on CentOS 7 or later/RedHat 8 or later), enter the following command to pull in Python 3.6 from RedHat Software Collections:

   ```
   # yum install -y \
     rh-python36-python \
     rh-python36-python-pip \
     rh-python36-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.

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
```

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.

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:
audit:
cluster:
monitoring:
```

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
        passwd: content-store-password
    cue_print:
        url: cue-print-host/newsgate-cf/
        user: cue-print-user
        passwd: cue-print-password
    dcx:
        url: dcx-endpoint-url
        user: dcx-user
        passwd: dcx-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.

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
```

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:
    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 6.

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 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 a 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.

`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:

`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.

These processor types 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:
    product-mapping:
    desk-mapping:
```

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

`system-id` (optional)
: May optionally 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 the value isn't configured here, it is read from the **`ZL_CUE_PRINT_EXTERNAL_SYSTEM_ID`** environment variable. If the environment variable isn't set either, then no external system ID is reported to CUE Print.

`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 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
  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.

`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: `native`)
The name of an existing import folder in the DC-X system, to which content will be uploaded. The following folder names (which exist in a standard DC-X installation) are recommended:

`native`
Use this folder when uploading binary assets.

`story`
Use this folder when uploading stories and storylines.

`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 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.

`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:

`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.

3.11 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 (see section 1.3). The configurations specified here 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.

```
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
```

`copyback` contains a **`fields`** property, 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).

3.12 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
```

3.13 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).

3.14 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 7.1).
- 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.

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.

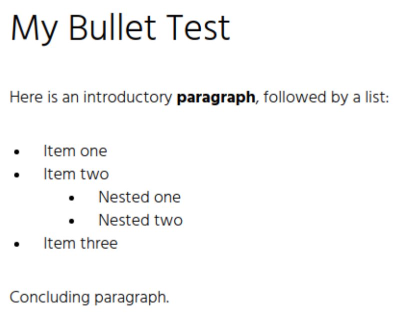
`classic`
: This folder contains templates for converting classic (rich text based) content items into CUE pseudo-storylines (`classic-to-storyline`) and vice-versa (`storyline-to-classic`).

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.

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": {
      "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": []
        }
      ]
    },
    "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": {
            "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": [],
                  "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
    }
  ],
  "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:

```
  <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..

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 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`** 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).

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 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:

```
{% 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 tags 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 endo of th 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:

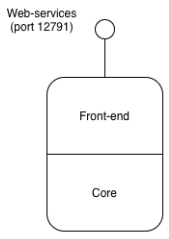
```
{% elif node.local_name == "italic" %}
```

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

5 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).

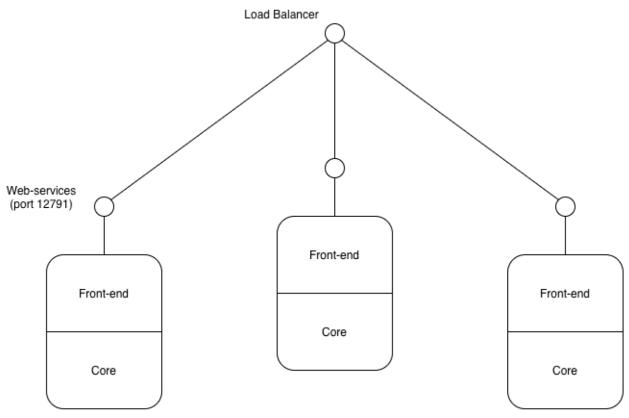


When running a CUE Zipline cluster, the front end web services should be load balanced, in order to distribute incoming client requests between the instances in the cluster.

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.

5.1 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 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).

5.1.1 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;
      ...
    }
  }
```

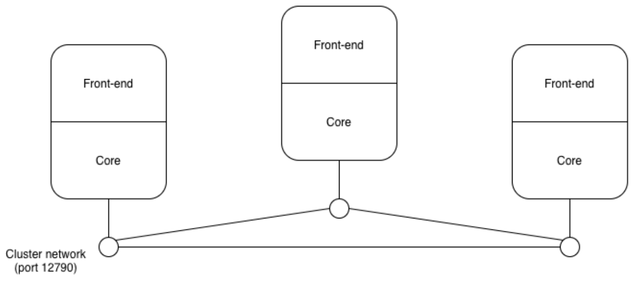
The **`upstream`** configuration should list the web service addresses for each instance in the CUE Zipline cluster. This address is the one configured in the section 3.4 configuration object. The default is port 12791 on the server the instance is running on.

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.

5.2 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 3.13 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.

5.2.1 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.

5.2.2 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.

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

5.2.3 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 3.13, 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:

```
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
```

6 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.

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.

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.

6.1 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.

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.

- 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.

7 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 3.14.

For information on how to request JSON reports, see section 7.1, and for information on the structure of the JSON reports, see section 7.2.

7.1 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 3.14).
- 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
  ```

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.

7.2 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.

7.3 Components

A report can contain the following components:

7.3.1 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.

7.3.2 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.

7.3.3 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.

`processing_time`
: The length of time spent processing each event, in seconds. Three different values are reported: average, minimum and maximum processing times.

8 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

8.1 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.

8.2 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.